GC27-6979-1
File No. S370-40

Systems

OS/VS Supervisor Services and Macro Instructions

OS/VS1 Release 2
OS/VS2 Release 1

IBM

Second Edition (September, 1972)

This is a major revision of, and supersedes, GC27-6979-0 and all associated SRL Newsletters.

This publication applies to release 2 of OS/VS1 and release 1 of OS/VS2 and to all subsequent releases unless otherwise indicated in new editions or technical newsletters. Changes are continually made to the specifications in this book; before using this book, consult the most recent edition of the IBM System/360 and System/370 Bibliography (GA22-6822) and the current SRL Newsletter for editions that are applicable and current.

Requests for copies of IBM publications should be made to your IBM representative or the IBM branch office serving your locality.

A form for reader's comments appears at the back of this publication. If the form has been removed, comments may be addressed to IBM Corporation, Programming Publications, Department 636, Neighborhood Road, Kingston, New York, 12401. All comments become the property of IBM.

Intended mainly for the programmer cod-
ıg in assembler language, this book
əscribes how to use the services of the
ıpervisor, the macro instructions used to
əquest these services, and the linkage
ɔnventions used by the control program to
:ovide these services.

This book is divided into two parts.
ırt I, "Supervisor Services", provides
ːplanations and aids for using the facili-
.es available through the supervisor by
əans of the macro instructions described
ı Part II, "Macro Instructions".

Part I is divided into seven topics.
ɔecific topics include:

Program Design: 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 are dependent on the completion of events in other tasks. This requires planned task synchronization. Planning is also required when more than one program will be using 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.

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

Section 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 lanaguge 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 information about these subjects are:

OS/VS1 Planning and Use Guide, GC24-5090
OS/VS2 Planning and Use Guide, GC28-0600
OS/VS Assembler Language, GC33-4010

When other IBM manuals are referred to in the text, only partial titles are given. The following is a list of the complete titles and order numbers of all manuals referred to in this book.

OS/VS
- Checkpoint/Restart, GC26-3784
- Data Management Macro Instructions, GC26-3793
- Data Management Services, GC26-3783
- JCL Reference, GC28-6704
- Linkage Editor and Loader, GC28-6451
- Service Aids, GC28-0633

OS/VS1
- Debugging Guide, GC28-6670

IBM System/370
- Principles of Operation, GA22-7000

CONTENTS

FIGURES

CHAPTER 1: INTRODUCTION TO 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 used 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. This requires planned task synchronization. Planning is also required when more than one program will be using a serially reusable resource.

Interruption, Termination, and Dumping: 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.

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

CHARACTERISTICS OF THE SYSTEM

Figure 1 gives a brief description of the control program options available for the operating systems that provide multiprogramming with virtual storage (OS/VS1 and OS/VS2). This figure does not attempt to cover all of the options available in the operating system; it only summarizes the options that affect the material covered in this book.

	VS1	VS2
Brief Description	Priority Scheduler, one (or, optionally, more than one) task per job step, 1 to 15 jobs processed concurrently	Priority Scheduler, one or more tasks per job step, 1 to 15+ jobs processed concurrently
Multiple Wait	Standard	Standard
Identify	Standard	Standard
System Log	Optional	Standard
Interval Timing	Standard	Standard
Multiple Console Support	Optional	Standard
Time Sharing	Not Available	Optional
Time Slicing	Optional	Standard
Dispatching	Dynamic (optional)	Heuristic

Figure 1. Summary of characteristics and available options

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.

LINKAGE CONVENTIONS

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 Hierarchy."

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 2). 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 immediately, you should load the address from register 1 into one of registers 2-12 or store the address in a fullword in your program.

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 3. 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."

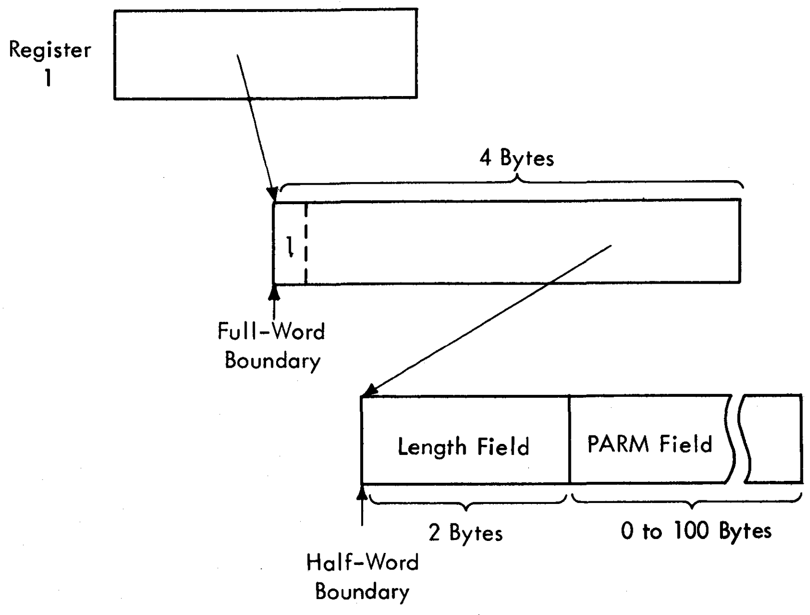


Figure 2. Acquiring PARM field information

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 3. Format of the save area

```
(A)  PROGNAME        SAVE(14,12)

(B)  PROGNAME        SAVE(5,10),T
```

Figure 4. SAVE macro instruction used to save (A) all registers but 13 and (B) registers 5-10, 14 and 15

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 4 illustrates uses of the SAVE macro instruction. The T operand (in B) specifies that the contents of registers 14 and 15 are to be saved.

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 5 and 6 show two methods of obtaining a save area and of saving all the registers, including the addresses of the two save areas. In Figure 5 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 instruction. 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.

In Figure 6, 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 address 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   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 5. Chaining save areas in a nonreenterable program

```
PROGNAME   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 6. 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 instruction.
- 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, place the entry address of your program in register 15.

VIRTUAL STORAGE CONSIDERATIONS

In a system with virtual storage you can distinguish between the address space used by a program (the program length) and the allocated real storage used for program execution. Real storage is allocated by the control program to meet realtime requirements. To use virtual storage effectively, you should consider the short-term demands on real storage and the time spent in allocating real storage (paging).

Paging

Virtual storage is sequentially addressed, beginning with location zero, and is mapped into real storage as it is referred to. The size of virtual storage is limited only by the addressing capability of the system, and the amount of auxiliary storage. Virtual storage is divided into a maximum of 256 segments. In VS1, each segment is further divided into 32 pages, each containing 2,048 bytes (2K) of addressable space. In VS2, each segment is divided into 16 pages, each containing 4,096 bytes (4K) of addressable space. The user and system are limited to a virtual storage capacity that is somewhat less than the 16 million bytes addressable with the 24-bit addressing method used by System/370. The limits are determined by the installation on the basis of such factors as real storage capacity, secondary storage capacity, and control program storage requirements. The control program controls both real and virtual storage. A page is the smallest unit of real storage that can be allocated. The process by which pages are transferred between real and external page storage is called paging.

An allocated virtual-storage page is always in one of three states: its content is on external page storage, its content is mapped into real storage, or it has no content and exists as address space only. Each time a page that is not in real storage is referred to, it must be paged into real storage.

The control program coordinates paging with the execution of other tasks. As paging time increases, system efficiency decreases. Since paging time is attributable in part to the way in which a task uses virtual storage, the programmer should consider paging when he designs a program.

Design Techniques

A task's paging rate is a principal factor in determining how efficiently the task is executed. The following techniques for designing a program will help minimize paging and thus help ensure more efficient operation.

- Code in page sections. It is useful to think of virtual storage as a "one-page overlay"; that is, only one page of virtual storage can be referred to without causing paging to occur.

- Keep seldom used subroutines, such as initialization and termination subroutines, separate from those which are used frequently. You can release whole pages of virtual storage associated with subroutines that are used only once. See "Relinquishing Virtual Storage" for details.

- Put frequently used subroutines in one page, preferably the page containing the most frequent callers. Similarly, group infrequently used or error routines together.

- Arrange data areas in a single page where possible. Avoid multiple-page lists and tables.

- Release pages for buffers as soon as they are all used.

- Keep frequently used programs or loops within programs in one page wherever possible.

- Keep dynamically changeable code in one page if possible. If a page has been changed it must be paged out before the real storage it occupies can be reused. Otherwise, the contents of the page can be overlaid.

- Arrange routines so that you avoid crossing page boundaries.

CHAPTER 3: SUBTASK CREATION AND CONTROL

One task is created 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 a job being executed. The benefits of a multiprogramming environment are still available even with only one task in the job step; work is still being performed for other jobs when your task is waiting for an event, such as an input operation, to occur.

The advantage in creating additional tasks within the job step is that more tasks are competing for control than the task in the job you are concerned with. When a wait condition occurs in one of your tasks, it is not necessarily a task from some other job 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 a job step) 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 and the current ability to use the 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, DE, and DCB operands. The use of these operands is discussed in "Program Management." Parameters can be passed to the subtask using the PARAM and VL operands, also described under "The LINK Macro Instruction." The only additional operands are those dealing with the priority of the subtask and the operands that provide for communication between tasks.

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.

TASK PRIORITY

Tasks compete for control on the basis of priority. When a task is created, it is assigned a priority, which can be revised upward or downward. It is also assigned a limit to its priority, a value equal to the highest priority the task can be assigned; this value is called the task's limit priority. The task's actual priority, the basis on which it competes for control, is called the task's dispatching priority.

A job step task can change its own dispatching priority but not its own limit priority. It can change both the dispatching and limit priorities of its subtasks, but cannot set the limit priority of a subtask higher than its own limit priority.

Priority of the Job Step Task

The limit priority of the job step task cannot be changed; it is always equal to the task's initial dispatching priority. You can specify initial dispatching priority through the DPRTY parameter of the EXEC statement.

The initial dispatching priority of the job step task is determined by the job priority. You either specify job priority through the PRTY parameter of the JOB statement or omit this parameter and allow the job priority to be determined by default. Job priority is used in selecting jobs for execution and in assigning input/output devices.

To specify job priority, code the parameter: PRTY=value

where value is the job priority, an integer from 0 to 13. If you do not specify dispatching priority for a job step, it is computed from the job priority as follows:

Dispatching Priority = (value x 16) + 11

Whether you specify dispatching priority or not, you cannot be absolutely certain of what a job step's initial dispatching priority will be. To achieve best results from the operating system, the operations staff may override specified job and dispatching priorities. Your program, therefore, cannot simply assume that the job step task will have a particular initial dispatching priority. To determine this priority, your program must issue the EXTRACT macro instruction, as described in the VS1 Planning and Use Guide and VS2 Supervisor Services and Macro Instructions for the System Programmer.

To summarize, the initial dispatching priority of the job step task can be established three ways:

- Indirectly through the PRTY parameter of the JOB statement
- By default when the PRTY parameter is omitted
- By the operations staff, overriding your own specifications

Which ever way it is established, the initial dispatching priority is always the limit priority for the job step task.

The job step task can lower its dispatching priority by use of the CHAP macro instruction. It can later use this macro instruction to revise its dispatching priority either upward or downward. It can never raise its dispatching priority above its initial dispatching (limit) priority.

Priority of Subtasks

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 operands of the ATTACH macro instruction. The LPMOD operand 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 operand 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. In that case, the limit priority 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.

If one or more subtasks must be completed before the originating task can execute beyond a certain point, the subtasks that must be completed should be assigned a priority that will prevent a long wait in the originating task.

Since tasks from other job steps are competing for CPU time, the priority initially established for the subtasks may be too high or too low to properly execute the job step. To correct this, the priorities of these subtasks can be changed by using the CHAP macro instruction. The EXTRACT macro instruction can be used to determine the priority of the current task and its subtasks. Note that each change of 16 in limit or dispatching priority is equivalent to a change of one in job priority.

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 the active task or a subtask is changed. The dispatching priority of the active task can be made less than the dispatching priority of another task waiting for control. If this occurs, the waiting task 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 the active task's subtasks. The 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.

DYNAMIC DISPATCHING (VS1)

Dynamic dispatching is an optional feature of the operating system which provides for the alteration of the dispatching priorities of selected tasks as they are being executed.

A dynamic dispatching algorithm calculates dispatching priorities so tasks can use system resources more efficiently. The algorithm not only alters the handling of each task as the task's characteristics change, but the algorithm also evaluates itself and alters itself, based on its effectiveness in handling the tasks under its control.

This algorithm distinguishes between I/O-bound tasks and CPU-bound tasks; I/O-bound tasks receive the higher priority. Initially, all tasks are designated I/O-bound; any new tasks entering the dynamically dispatched group are also designated I/O-bound. As each task is dispatched, its activity is monitored for a predetermined time interval. At the end of this time interval each task is designated I/O-bound or CPU-bound.

Dynamic dispatching assumes that the task most likely to be I/O-bound the next time it is dispatched is the task that was I/O-bound the last time it was dispatched.

Each time a task in the dynamically dispatched group relinquishes control of the CPU, its relative position within the task queue may change as shown in Figure 6A.

Original Task Status	Reason for Loss of CPU Control	New Task Status	Action Taken
I/O	Voluntary Surrender	Unchanged	Search down I/O-queue for next task to dispatch
I/O	Time Interval Ended	CPU	Move task to head of CPU subgroup and search down I/O-queue from old location of task
I/O	Preemption for Another Task	Unchanged	Dispatch preempting task
CPU	Voluntary Surrender and no Interval Expiration	I/O	Move task to bottom of I/O subgroup and search down CPU-queue from old location of task
CPU	Time Interval Ended	Unchanged	Move task to bottom of CPU subgroup and search down CPU-queue from old location of task
CPU	Preemption for Another Task	Unchanged	Move task to bottom of CPU subgroup and dispatch pre-empting task
CPU	Voluntary Surrender and Interval Expiration	Unchanged	Move task to bottom of CPU subgroup and search down CPU-queue from old location of task

Figure 6A. Status change as a result of loss of CPU control

I/O-bound tasks that remain I/O-bound tend to migrate higher in the queue as other I/O-bound tasks change status and join the CPU-bound task queue. A task that changes from I/O-bound to CPU-bound receives the highest dispatching priority within the CPU-bound group. In contrast, a CPU-bound task that switches to the I/O-bound group has the lowest priority within the I/O-bound group. This mechanism aids in making a finer distinction between tasks with relatively constant I/O activity and those tasks characterized by many status changes.

The cyclic movement of the CPU-bound tasks ensures that they will all share in any available CPU time. Therefore, these tasks have increased chances to change their status; potential I/O tasks are not locked in at the bottom of the CPU-bound queue.

Note: CPU-bound tasks that voluntarily surrender control of the CPU are examined between wait states. If the task's time interval expires, a bit is posted in the task's TCB to indicate the expiration. When the task issues a WAIT, this bit in the TCB is checked. If the time interval has expired, the bit is reset, and the status of the task remains CPU-bound.

If the time interval has not expired since the previous WAIT, the status of the task is changed to I/O-bound.

The user must specify the desired ratio of CPU-bound tasks to I/O-bound tasks. The algorithm checks this ratio, and when there are too many CPU-bound tasks in the system, the time interval is lengthened. If there are too many I/O-bound tasks in the system, the time interval is decreased.

Dynamic dispatching is a SYSGEN option. The CTRLPROG macro will have parameters for specifying the following information to the control program.

- First and last partitions in the dynamic dispatching group (partitions must be contiguous). There is no default value for this parameter.
- Desired ratio of CPU-bound tasks to I/O-bound tasks. This value is specified in hundredths (for example, a ratio of 5 to 1 is specified as 500; a ratio of 1 to 2 is specified as 50). The range is 1 to infinity. The default value is 2 CPU-bound tasks to 1 I/O-bound task.
- The low limit for the time interval. The default is 20 milliseconds.
- The initial delta by which the time interval will be modified. The default value is 5 milliseconds.
- The length of the statistics interval. The default is 10 seconds.

Any tasks that are created by an ATTACH issued by a task in a dynamic dispatching partition are also part of the dynamic dispatch group. The LPMOD and DPMOD parameters are ignored for these subtasks. A task cannot issue a CHAP macro instruction to move itself into or out of a dynamic dispatching partition; the CHAP instruction results in a NOP.

Time slicing and dynamic dispatching may not exist within the same partition, but they may coexist within the system.

TIME SLICING

Time slicing is a feature of the operating system which enables tasks that are members of a time-slicing group to share control of the CPU. When a member of the time-slicing group has been active for a certain length of time, it is interrupted, and control is given to another member of the group. In this way, all member tasks are given equal slices of CPU time; no task can use the CPU to the exclusion of all others.

Any task or subtask is a member of a time-slicing group if its dispatching priority is within the range of the dispatching priorities assigned to partitions designated for time slicing. A task or subtask can use the CHAP macro instruction to become a member of the time-slicing group if its limit priority is equal to or greater than the lowest dispatching priority of the time-slicing group. Also, an originating task can use the ATTACH or CHAP macro instruction to designate a subtask as a member of the time-slicing group if the limit priority of the originating task is equal to or greater than the lowest dispatching priority of the time-slicing group.

Partition Number	Highest Dispatching	Lowest Dispatching
0	251	241
1	240	230
2	229	219
3	218	208
4	207	197
5	196	186
6	185	175
7	174	164
8	163	153
9	152	142
10	141	131
11	130	120
12	119	109
13	108	98
14	97	87
15	86	76
16	75	65
17	64	54
18	53	43
19	42	32
20	31	21
21	20	10
22	9	1
23-n	0	0
Reduce the dispatching priorities by one for each of the following functions included in system: system log, system management facility, I/O recovery management support.		

Figure 7. Dispatching priorities of partitions

VS1 Systems

Each partition has a range of eleven dispatching priorities assigned to it. The range of dispatching priorities for a time-slicing group is from the highest dispatching priority of the highest priority partition within the group to the lowest dispatching priority for the lowest priority partition within the group (see Figure 7). For example, if partitions 6 through 8 were assigned to the time-slicing group, any task or subtask whose dispatching priority fell within the range 185-153 would be a member of the time-slicing group. If the system log and system management facility functions were included in the system, the range of time-slicing dispatching priorities would be 183-151.

VS2 Systems

At system generation, your installation designates certain job priorities for time slicing. Your tasks are members of the time-slicing group if their dispatching priorities correspond to these job priorities. For example, if job priorities 8 and 9 are designated, tasks are members of the time-slicing group when their dispatching priorities can be computed as follows:

For job priority 8,
Dispatching Priority = (8 x 16) + 11 = 139

For job priority 9,
Dispatching Priority = (9 x 16) + 11 = 155

In this example, tasks with priorities 139 and 155 are members of the time-slicing group. Note that time slicing applies only to ready tasks with the highest priority; a task with priority 155 would not be interrupted to give control to a task with priority 139.

Time slicing is important chiefly in real-time applications, but it affects the use of the ATTACH and CHAP macro instructions by all tasks in the system. These macro instructions determine task priorities, and therefore determine membership in the time-slicing group. In using these macro instructions, you must consider carefully the priorities for which time slicing is performed at your installation. Using the ATTACH and the CHAP macro instructions can affect dispatching priorities, as discussed above.

Consider again the example in which time slicing is performed for job priorities 8 and 9. If a job-step task has an initial dispatching priority of 139, it is initially a member of the time-slicing group. If it lowers its priority, it is no longer a member of the group; if it attaches a subtask, the subtask is a member only if it is assigned a dispatching priority of 139 (the limit priority of the job-step task).

If another job-step task is assigned an initial dispatching priority greater than 155, it is not initially a member of the time-slicing group. However, it can create lower priority subtasks that are members of the time-slicing group, and can itself become a member by lowering its own dispatching priority to 155 or 139. Note that careless use of the ATTACH and CHAP macro instructions could result in a task's becoming a member of the time-slicing group when time slicing is not actually intended.

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 8. The horizontal lines in Figure 8 separate originating tasks and subtasks; they have no bearing on task priority. Tasks A, A1, A2, A2a, B, B1, and B1a are all <u>subtasks</u> of the job-step task; tasks A1, A2, and A2a are subtasks of task A. Tasks A2a and B1a are the lowest <u>level</u> tasks in the job step. Although task B1 is at the same <u>level</u> as tasks A1 and A2, it is not considered a subtask of task A.

Task A is the <u>originating</u> 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.

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 operands, the ECB and ETXR operands, are provided in the ATTACH macro instruction to assist in communication between a subtask and the originating task. These operands are used to indicate the normal or abnormal termination of a subtask to the originating task. If the ECB or ETXR operand, 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

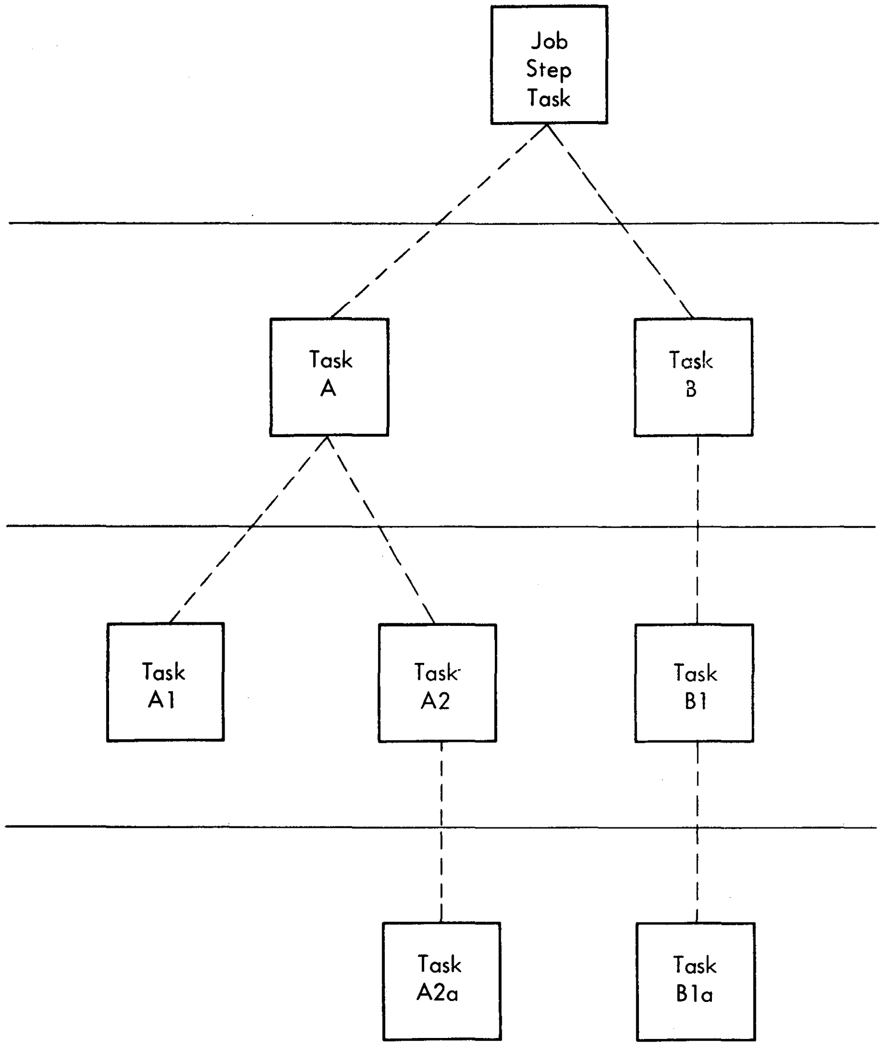


Figure 8. Levels of tasks in a job step

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 operand 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 can be given control even though the originating task is in the wait condition. When the end-of-task routine returns control to the control program, the originating task remains in the wait condition if the event control block has not been posted. 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 operand 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 operand is specified in the ATTACH macro instruction, the task control block for the subtask is removed from the system when the subtask is terminated. It's 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.

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 9.

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

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.

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

Figure 9. Characteristics of load modules

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 VS 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 which program 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 10. 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.

An example of passing a parameter list is shown in Figure 11. Early in the routine the contents of register 1 (that is, the address of the

```
               ...
               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 10. Passing control in a simple structure

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

Figure 11. Passing control with a parameter list

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.

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 12 and 13 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 12. Passing control with return

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

Figure 13. Passing control with CALL

Use of an inline parameter list and an answer area is also illustrated in Figure 12. The address of the external entry point is loaded into register 15 in the usual manner. A branch and link instruction is then used to branch around the parameter list and to load register 1 with the address of the parameter list. An inline parameter list such as the one shown in Figure 12 is convenient when you are debugging because the parameters involved are located in the listing (or the dump) at the point they are used, instead of at the end of the listing or dump. Note that the 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.

The CALL macro instruction in Figure 13 provides the same functions as the instructions in Figure 12. When the CALL macro instruction is expanded, the operands 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 causes the load module with the entry point NEXT to be automatically 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 13, contains only A-type address constants. A variation on this type of parameter list results from the following coding:

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

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

Analyzing the Return

When 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 14 could be used to analyze the results. Many times, however, the results and the alternatives are more complicated, and a branching table, such as shown in Figure 15, could be used to pass control to the proper routine.

Note: Explicit tests are required to ensure that the return code value does not exceed the branch table size. Never assume that no expansion of return codes will occur.

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

Figure 14. Test for normal return

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

Figure 15. Return code test using branching table

How Control is Returned

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

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

The return address is the address originally passed in register 14; 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 16 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 16 loads register 13 with the address of the save area this control section used and 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.

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.

```
           ...
           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 16. Establishing a return code

```
           ...
           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 17. Using the RETURN macro instruction

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

Figure 18. RETURN macro instruction with flag

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

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

Return to the Control Program

The discussion in the preceding paragraphs has covered passing control within one load module, and has been based on the assumption that the load module was brought into virtual storage because of the program name specified in the EXEC statement. The control program established only one task to be performed for the job step. When the logical end of the program is reached, control passes to the return address passed (in register 14) to the first control section in the 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.

- In VS2 systems, unique task libraries may be established by using the TASKLIB operand 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 operand 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 resident reenterable module area, (VS1), or the link pack area (VS2). The contents of these areas are determined during the nucleus initialization process and will vary depending on the requirements of your installation. In an operating system with VS2, the link pack area contains frequently used, 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. If TSO is in the system and is started, it extends the link pack area. The resident reenterable module area can contain user-written modules and the loader, discussed in the Linkage Editor and Loader publication, and all reenterable graphics subroutine package (GSP) modules.

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 resident access module (RAM) list (VS1), the task's load list, the link pack area (VS2), and the resident reenterable module area (VS1). 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 can not 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 whether the system is VS1 or VS2 and on the operands used in the macro instruction requesting the load module. The operands that define the order of the search are EP, EPLOC, DE, and DCB. The EP, EPLOC, and DE operands 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 operand is used to indicate the address of the data control block for the library containing the load module, and is optional. Omitting the DCB operand or using the DCB operand with an address of zero specifies the data control block for the link library or the job or step library.

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

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

When used without the DCB operand, the EP and EPLOC operands provide the easiest method of requesting a load module from the link, job, or step library. In a system with VS2, 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. In VS1, the data sets that make up these libraries are searched in the order of their DD statements.

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

VS1	VS2
The job pack area (user storage) is searched.	The job pack area of the region is searched for an available copy
The step library or the job library (if any) is searched. If both libraries are specified, the job library is not searched.	The requesting task's task library and all the unique task libraries of its antecedent tasks are searched.
The link pack area or the resident reenterable load module area is searched (optional).	The step library is searched; if there is no step library, the job library (if any) is searched.
The link library is searched.	The link pack area is searched.
	The link library is searched.

Figure 19. Search for module, EP or EPLOC operand with DCB=0 or DCB operand omitted

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 operand must be coded in the macro instruction.

This is not the case for task libraries. Extreme coution 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 operand, this time in conjunction with the DCB operand. You specify the address of the data control block for the private library in the DCB operand. The order of the search for EP or EPLOC with the DCB operand is shown in Figure 20.

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 operand requires more work than the EP and EPLOC operands, but it can reduce the amount of time spent searching for a load module. Before you can use this operand, 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.

VS1	VS2
The partition is searched.	The job pack area of the region is searched for an available copy.
The resident reenterable load module area is searched (optional).	The specified library is searched.
The specified library is searched.	The link pack area is searched.
	The link library is searched.

Figure 20. Search for module, EP or EPLOC operands with DCB operand specifying private library

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 operand, you provide the address of the directory entry and code or omit the DCB operand to indicate the same library specified in the BLDL macro instruction. The order of the search when the DE operand is used is shown in Figure 21 for the link, job, step, and private libraries.

The preceding discussion of the search is based on the premise that the entry name you specified is the member name. In an operating system with VS1, the same search results from specifying an alias rather than a member name. When you are using VS2, 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.

As the discussion of the search indicates, you should choose the operands 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 operands that eliminate as many of these unnecessary searches as possible, as indicated in Figures 19, 20, and 21. Examples of the use of these figures are shown in the following discussion of passing control.

Using an Existing Copy

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

Operating System with VS1: The LOAD macro instruction permits all tasks in a partition to share the same copy of a reenterable module requested by a previous LOAD macro instruction. If the reenterable module is later requested by a LINK, XCTL, or ATTACH macro instruction and a previous request is still active, the existing copy of the module is used.

VS1	VS2
Directory Entry Indicates Link Library and DCB=0 or DCB Operand Omitted	
The partition is searched.	The job pack area for the region is searched for an available copy.
The resident reenterable load module area is searched (optional).	The link pack area is searched.
The module is obtained from the link library.	The module is obtained from the link library.
Directory Entry Indicates Job, Step, or Task Library and DCB=0 or DCB Operand Omitted	
The job pack area for the partition is searched for an available copy.	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.	The module is obtained from the step library; if there is no step library, the module is obtained from the job library.
DCB Operand Indicates Private Library	
The job pack area for the partition is searched for an available copy.	The job pack area for the region is searched for an available copy.
The module is obtained from the specified private library.	The module is obtained from the specified private library.

Figure 21. Search for module using DE operand

Operating System with VS2: 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 instrucion 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 partition/region or job pack area if it was not preloaded into the resident reenterable module area, the resident access method area, or 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 partition/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 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 responsibility count for a copy in a job pack area reaches zero, the virtual storage area containing the copy is made available; the copy is never reused after the responsibility count established by LOAD macro instructions reaches zero.

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 a 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 will 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 operands. 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 responsibility count for a load module when control is passed using the LINK macro instruction. This is a separate responsibility 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 22 and 23 show the coding of a LINK macro instruction used to pass control to an entry point in a load module. In Figure 22, the load module is from the link, job, or step library; in Figure 23, the module is from a private library. Except for the method used to pass control, this example is similar to Figures 12 and 13. 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 operand is chosen, since the search begins with the job pack area and the appropriate library as shown in Figure 19.

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

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

Figure 22. Use of the LINK macro instruction with the job or link library

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

Figure 23. Use of the LINK macro instruction with a private library

```
            BLDL    0,LISTADDR
            ...
            DS      0H              List description field:
 LISTADDR   DC      H'01'              Number of list entries
            DC      H'58'              Length of each entry
 NAMEADDR   DC      CL8'NEXT'       Member name
            DS      25H             Area required for directory information
```

Figure 24. Use of the BLDL macro instruction

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

Figure 25. The LINK macro instruction with a DE operand

The first operand of the BLDL macro instruction is a zero, which indicates that the directory entry is on the link or job library. The second operand is the address in virtual storage of the list description field for the directory entry. The first two bytes at LISTADDR indicate the number of directory entries in the list; the second two bytes 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 58 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 25 can now be written. Note that the DE operand 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 to register 0 and the length 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 26 shows how this could happen. Control is given to load module A, which passes control to load module B (step 1) using a LOAD macro instruction and a branch and link instruction. Register 14 at this time contains the address of the instruction following the branch and link. Load module B then is executed, independently of how control was passed, and issues an XCTL macro instruction when it is finished (step 2) to pass control to load module C. The control program, knowing only of load module A, lowers the responsibility count of A by one, resulting in its deletion. Load module C is executed and returns to the address which used to follow the branch and link instruction. Step 3 of Figure 26 indicates the result.

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

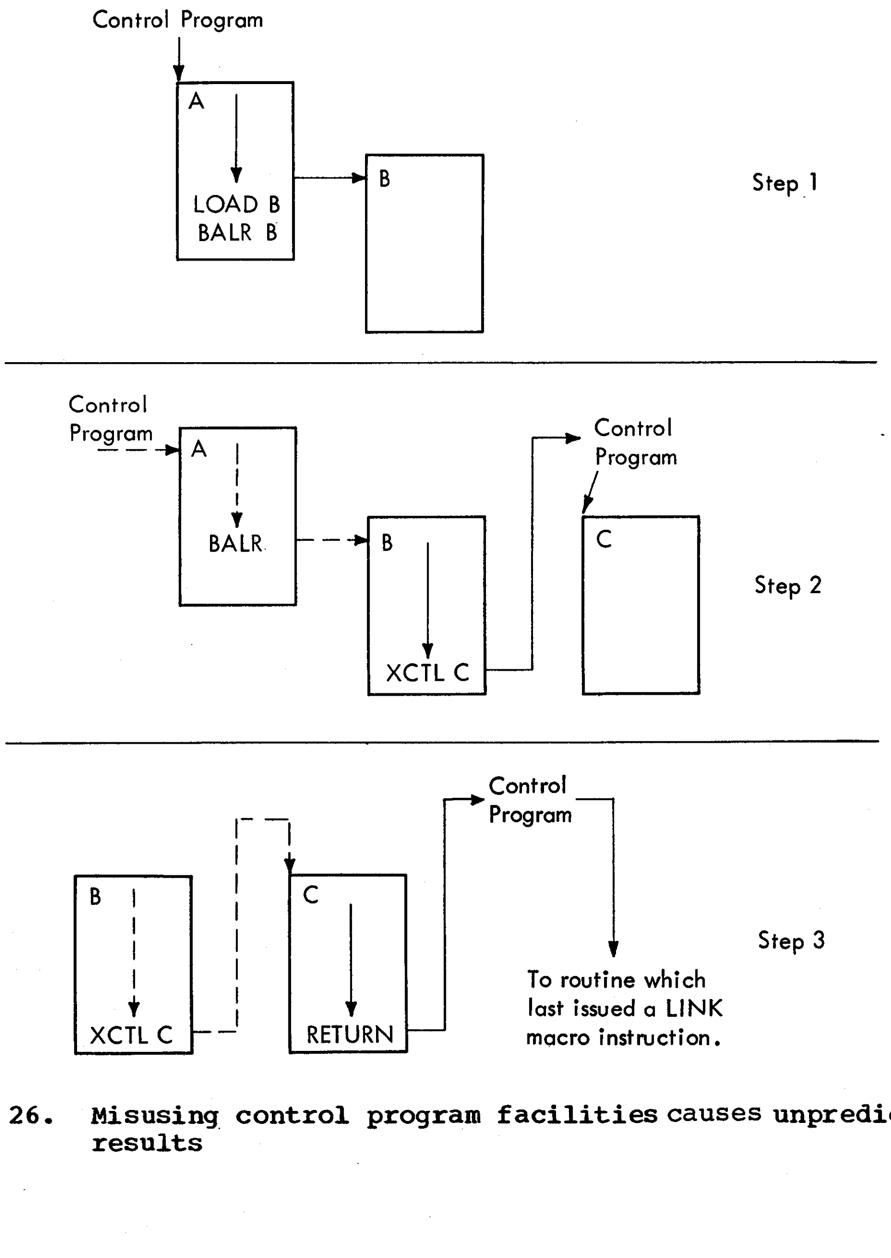


Figure 26. Misusing control program facilities causes unpredictable results

ADDITIONAL ENTRY POINTS

Through the use of linkage editor facilities you can specify as many as 17 different names (a member name and 16 aliases) and associated entry points within a load module. It is only through the use of the member name or the aliases that a copy of the load module can be brought into virtual storage. Once a copy has been brought into virtual storage, however, additional entry points can be provided for the load module, subject to 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. In VS1, an IDENTIFY macro instruction cannot be issued when the load module is given control at an entry point that was added by an IDENTIFY macro instruction.

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, the job pack area, and the partition/region of the job step 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.

In VS1, the added entry point can be used only in an ATTACH macro instruction. The added entry point is available for as long as the copy is retained in virtual storage. Proper task synchronization is required when using an added entry point in the performance of a task which has not directly requested the associated copy of the load module; the load module may otherwise be deleted before the use is complete. The added entry point is treated as an entry point to a reenterable load module by the control program, regardless of the actual module attributes of the load module. You must guard against reuse of nonreusable coding.

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. The dump program uses the calling sequence identifier and the entry point identifier as shown in the VS1 Debugging Guide.

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.

CHAPTER 5: RESOURCE CONTROL

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 macro instruction is used to indicate that a task cannot proceed until one or more events have occurred. An event control block is used with the WAIT and POST macro instructions; it is a fullword on a fullword boundary, as shown in Figure 27.

An event control block is also used when the ECB operand is coded in an ATTACH macro instruction. In this case the control program issues the POST macro instruction for the event (subtask termination). Either the 24-bit (bits 8 to 31) return code in register 15 (if the task completed normally) or the completion code specified in the ABEND macro instruction (if the task was abnormally terminated) is placed in the event control block as shown in Figure 27. The originating task can issue a WAIT 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).

When an event control block is originally created, bits 0 (wait bit) and 1 (post bit) must be set to zero. An event control block can be reused; if it is reused, bits 0 and 1 must be set to zero before either the WAIT or POST macro instruction can be used. 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.

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 an event control block at one 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.

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

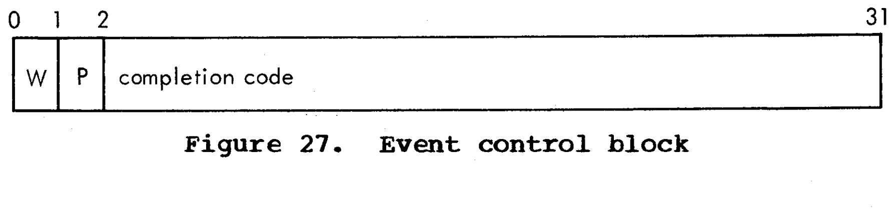


Figure 27. Event control block

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. The control program determines the status of the resource, and either grants the request by returning control to the active task or delays assignment of control by placing the active task in the wait condition. 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. 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 and rname 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 and rname to represent the same resource. The control program treats requests having different qname and rname combinations as requests for different resources. 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 can assign the qname and rname combination. You should, in this case, code the STEP operand 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 job step identifier to the rname so that duplicate qname and rname combinations are not used unintentionally in different job steps. If the resource is available to any job step in the system, the qname and rname combination must be agreed upon by all users. The SYSTEM operand 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 you might accidentally duplicate one of these.

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 an exclusive request, no other task will be given simultaneous control of the resource. 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 and rname 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 and rname combination for which a list exists; if no list exists for that qname and rname 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 28 shows the status of a list built for a very popular qname and rname 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 28, 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.

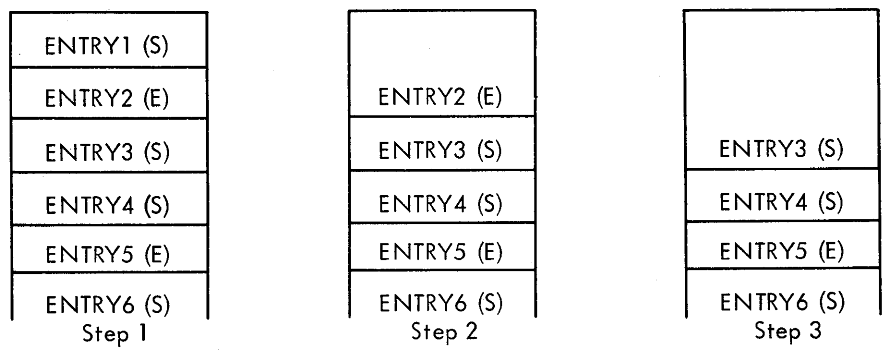


Figure 28. ENG macro instruction processing

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 28, 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 28, 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 abnormal termination of the task. You must 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, 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 operand in the ENQ and DEQ macro instructions. The RET operand (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 and rname 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 12 means the resource is permanently unavailable. 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 12 means the resource is permanently unavailable. 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. A return code of 12 means the resource is permanently unavailable. Although this is an error condition, control is returned tc the user.

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 12 means the resource is permanently unavailable. 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 coe of 8 indicates that the task did not have an entry for the resource (although the task may be in the wait condition because of a request 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 29. 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.

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.

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

Figure 29. Interlock condition

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

Figure 30. Two requests for two resources

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

Figure 31. One request for two resources

- Share resources as much as possible. If the requests in the lists shown in Figure 29 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 30, the one ENQ macro instruction shown in Figure 31 could be coded. If all requests were made in this manner, the interlock shown in Figure 29 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 be written in the same manner to release the entire set of resources at once.

- If the use of one resource always depends on the use of a second resouce, 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 operand, 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.

PROGRAM INTERRUPTION PROCESSING

Some conditions encountered in a program cause a program interruption. These conditions include incorrect operands and operand 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 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 the SPIE macro instruction produces a control program parameter list, called a program interruption control area (PICA). The PICA, shown in Figure 32, contains the new program mask for the interruption types that can be disabled, the address of the exit routine to be given control, and a code for interruption types (exceptions) specified in the SPIE macro instruction.

A program that issues a SPIE macro instruction must restore the PICA that was in effect when control was received. It must do so before it returns control to the calling program, or transfers control to another program by issuing an XCTL macro instruction. When the SPIE macro instruction is issued, the control program returns the address of the

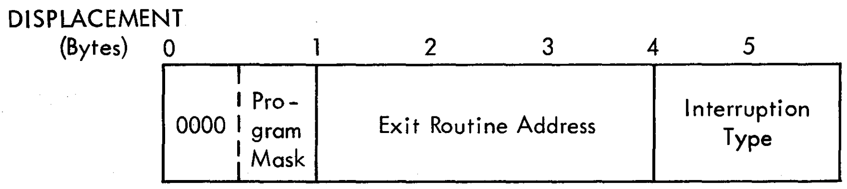


Figure 32. Program interruption control area

previous PICA in register 1. The control program returns zero in register 1 when there is no previous PICA, that is, when no SPIE macro instruction has been issued earlier in performance of the task.

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

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 and, in VS1, a 32-byte work area in the protected queue area for the program check handler. This program interruption element is used each time a SPIE macro instruction is issued during the performance of the task and contains the information shown in Figure 34.

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.

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

Figure 33. Using the SPIE macro instruction

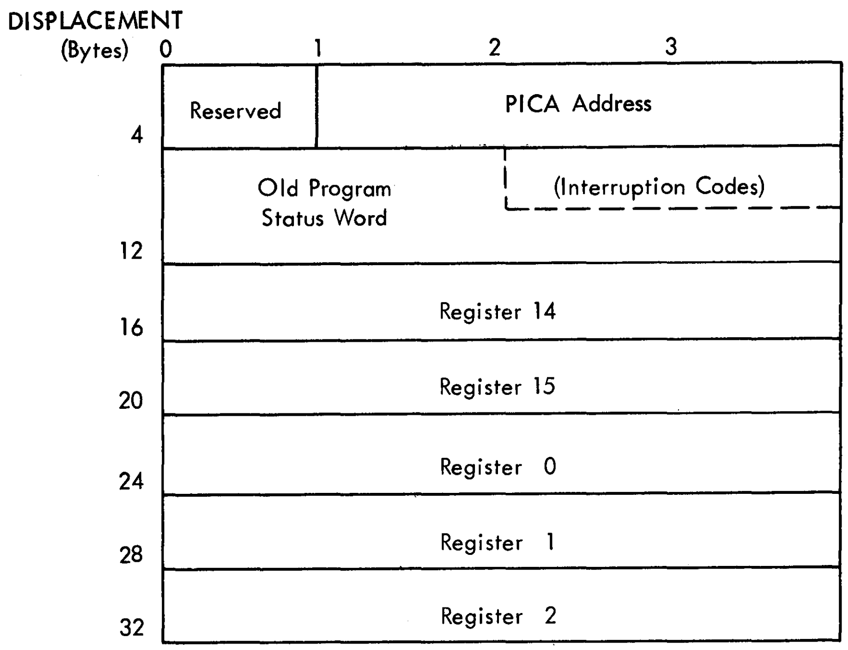


Figure 34. Program interruption element

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 35 is an example of one of these. The routine shown in Figure 35 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

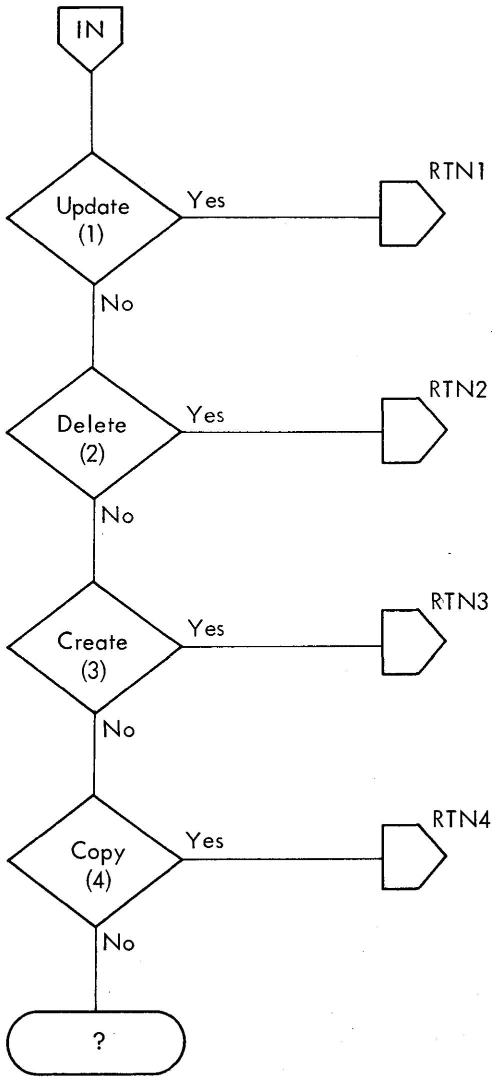


Figure 35. Detecting an abnormal condition

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.

An alternative to this procedure is to pass control to the control program abnormal termination routine by issuing an ABEND macro instruction. This alternative is simpler, but it offers less opportunity for error recovery and continued processing unless a STAE macro instruction, specifying a STAE exit routine address, is issued to override the ABEND. The use of STAE is discussed in the Planning and Use Guide. The abnormal termination facilities available through the use of the ABEND macro instruction are discussed below; an explanation of the facility to intercept abnormal termination through the STAE macro instruction is presented following the ABEND discussion.

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 operand 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 operand) 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 most frequent cause of this is that the subtask does not have sufficient virtual storage for ABEND's processing. ABEND "steals" virtual storage allocated to the job step task and needed by it to continue normal processing. The abnormal termination routine works in the same manner whether it is given control from the control program or a problem program.

When a task is abnormally terminated, the control program performs the following functions:

- Lowers the responsibility counts for the load modules brought into virtual storage during the performance of the task.
- Releases the virtual storage subpools owned by the tasks.
- Cancels the time interval if one had been established for the task.
- Issues a CLOSE macro instruction for any data control blocks that were opened during the performance of the task.
- Purges any outstanding input or output requests.
- Cancels any requests for operator replies made using a WTOR macro instruction.
- Cancels any requests for resources made using an ENQ macro instruction.

If the job step is not to be terminated, the following action is taken:

- The abnormal termination functions listed above are performed, starting with the lowest level task, for each of the subtasks of the task that was active when the ABEND macro instruction was issued.

- The 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 operand 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 operand 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 operands were 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 action is taken:

- The abnormal termination functions listed above are performed, starting with the lowest level task, for all tasks in the job step. All virtual storage belonging to the job step is released. None of the end-of-task exit routines are given control.

- The 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_.

DUMPING SERVICES

There are three types of storage dumps produced by the operating system:

- A dump obtained through use of the DUMP operand in the ABEND macro instruction.

- A dump obtained through use of the SNAP macro instruction.

- An SVC dump, produced in the event of a failure by a system routine.

You can request a dump by using the ABEND or SNAP macro instruction. You cannot request the SVC dump -- it is produced automatically by the system whenever a failure occurs in a system routine.

ABEND AND SNAP DUMPS

When the dump is requested using an ABEND macro instruction, no further processing is performed for the active task; use of the SNAP macro instruction allows the task to continue after the completion of the dump. The control program usually request a dump for you when it issues an ABEND macro instruction.

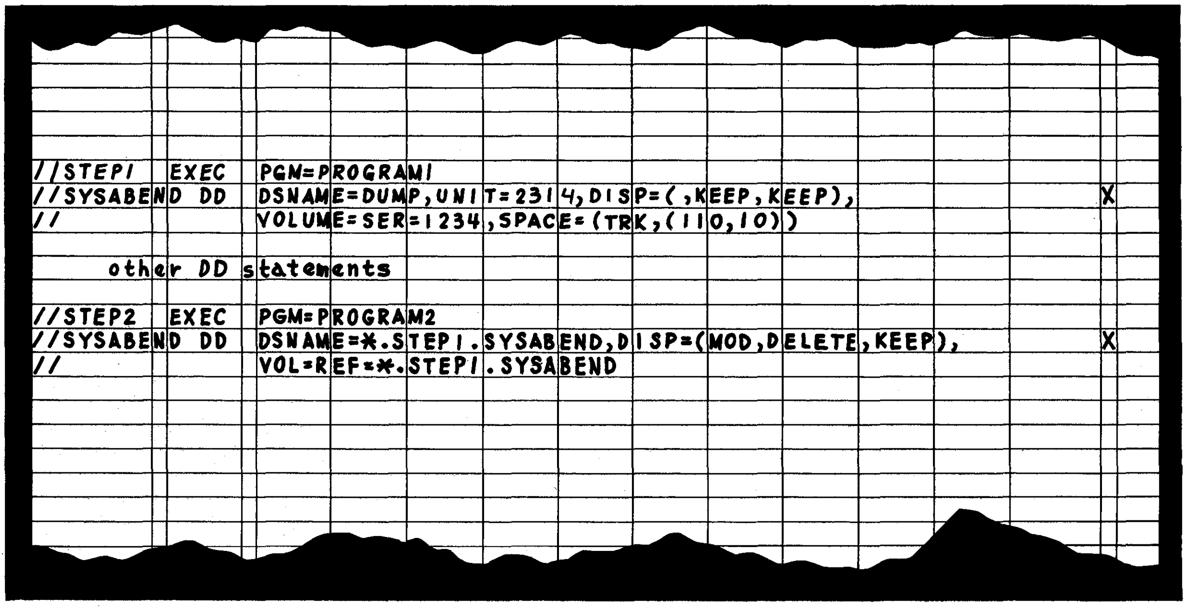
The data set containing the dump can reside on any device which is supported by the basic sequential access method (BSAM). The dump is placed in the data set described by the DD statement you provide. If a printer is selected the dump is printed immediately. However, if a direct access or tape device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

The format of the dump is shown in the publication VS1 Debugging Guide. The entire dump shown in that publication is provided in an abnormal termination dump if a DD statement with a DD name of SYSABEND is provided; only the problem program areas and system control blocks associated with the problem program are dumped if a DD statement with a DD name of SYSUDUMP is provided. Use of the SNAP macro instruction allows you to request only selected portions of the entire dump for any task in the job step; the format of the portions selected is the same as the format of the same portions of an abnormal termination dump.

When an abnormal termination dump is requested, the entire dump is provided for the active task, along with a dump of the control blocks and save area for each of the higher level tasks which are predecessors of the active task being terminated and for each of the subtasks of the active task. The control program dump routine uses the addresses you stored in words 2 and 3 of each save area to follow the chain of save areas provided by each calling program in each task. If an ABEND macro instruction was issued when task B1 (Figure 8) was active, for example, a complete dump would be provided for task B1. The control blocks and save areas for task B, task B1a, and the job step task would also be provided in separate dumps.

To get a dump:

- You must provide a DD statement for each job step in which a dump is requested. For an abnormal termination dump, the DD name must be SYSABEND or SYSUDUMP; for a SNAP macro instruction dump, the DD name must be any name except SYSABEND or SYSUDUMP. Figure 36 shows a set of job steps that include DD statements for ABEND dump data sets.



```
//STEP1    EXEC   PGM=PROGRAM1
//SYSABEND DD     DSNAME=DUMP,UNIT=2314,DISP=(,KEEP,KEEP),                 X
//                VOLUME=SER=1234,SPACE=(TRK,(110,10))

      other DD statements

//STEP2    EXEC   PGM=PROGRAM2
//SYSABEND DD     DSNAME=*.STEP1.SYSABEND,DISP=(MOD,DELETE,KEEP),          X
//                VOL=REF=*.STEP1.SYSABEND
```

Figure 36. Sample DD statements for an ABEND dump

- 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 yourprogram is to be processed by the loader, you should also issue a CLOSE macro instruction for the SNAP data control block.

INDICATIVE DUMP (VS1)

You can obtain an indicative dump, as shown in _VS1 Debugging Guide_. This dump is provided in response to a request for an abnormal termination dump when either you did not provide a DD statement with the DD name SYSABEND or SYSUDUMP, or the control program entry for that DD statement was destroyed. The indicative dump is printed on the system output device.

SVC DUMP

If a system routine fails, the system automatically supplies a dump of the contents of virtual storage. This dump, called the SVC dump, provides diagnostic information. The system writes the dump in the system data set SYS1.DUMP or in a tape volume at the device designated when the operating system was initially loaded. Use the HMDPRDMP service aid program to obtain a printout of the dump. A description of HMDPRDMP and the dump formats appear in the _Service Aids_ publication. For guidance in using the dump, refer to the _VS Debugging Guides_.

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 virtual storage request is satisfied by allocating to the active task 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 in VS2 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 partition, and because you only make one request, the amount of control program overhead is less.

Types of Explicit Requests

There are three 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 operand field of the GETMAIN macro instruction, has certain advantages, depending on the requirements of your program. The last two 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 (R): 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 (E): 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 (V): 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.

In addition to the above methods of requesting virtual storage, you can designate the request as conditional or unconditional. (A register type request is always 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 36. 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.

A conditional request for a single element of storage with a length of 16,000 bytes is requested in Figure 37. 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.

```
          ...
          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 allo-
                                            cated, proceed
          DELETE   EP=REENTMOD              If not,
          GETMAIN  VU,LA=SIZES,A=ANSWADD    Try to get smaller amount
                                            in 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        '8000'                  Min. size for procedure 1
SIZES     DC       F'16000'                 Min. size to proceed
                                            size of area for maximum
                                            efficiency
ANSWADD   DC       F'0'                     Address of allocated area
          DC       F'0'                     Size of allocated area
```

Figure 37. Using the GETMAIN macro instruction

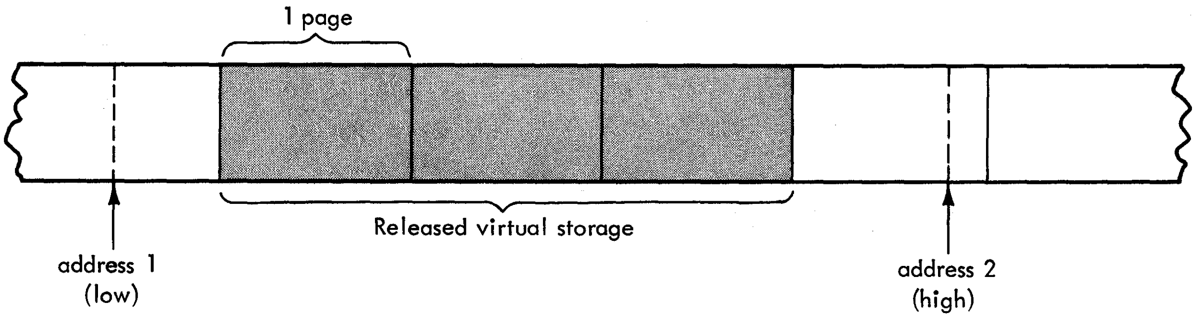


Figure 38. Releasing virtual storage

Relinquishing 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 an area of virtual storage within your program no longer has meaningful or significant contents, you can make this storage available by issuing a PGRLSE macro instruction. The PGRLSE macro makes all real and external page storage wholly associated with the area of virtual address space specified available as shown in Figure 38. The address space remains intact, 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 address space without the necessity of paging operations, PGRLSE may improve operating efficiency.

When you issue a FREEMAIN macro instruction, FREEMAIN does the equivalent of PGRLSE for any resulting free page.

Subpool Handling (In VS1 Systems)

Although subpools are not created in VS1 systems, it is convenient to call the partition itself "subpool 0." That is, all virtual storage available to the user in a partition is shared by all tasks active in that partition.

User programs may request virtual storage from the partition by specifying any subpool number from 0 to 127 or by specifying no number at all. Implied requests for storage, initiated when the user executes an ATTACH, LINK, LOAD, or XCTL macro instruction, are recorded by the control program in order for the storage to be freed during termination.

Subpool Handling (In VS2 Systems)

In an operating system with VS2, 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. 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.

Note: If virtual storage is allocated to a subtask by the user program while the system is being executed in the supervisor state or with a protection key of 0, no other task should free that virtual storage. If some other task does free that virtual storage, you get unpredictable results.

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

Figure 39 is a simplified view of a virtual-storage region containing four 4096-byte blocks of storage. All the requests are for virtual storage from subpool 0. The first request from some task in the job step is for 1008 bytes; the request is satisfied from the block shown as Block A in the figure. The second request, for 4000 bytes, is too large to be satisfied from the unused portion of Block A, so the control program assigns the next available block, Block B, to subpool 0, and allocates 4000 bytes from Block B to the active task. A third request is then received, this time for 2000 bytes. There is 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. 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

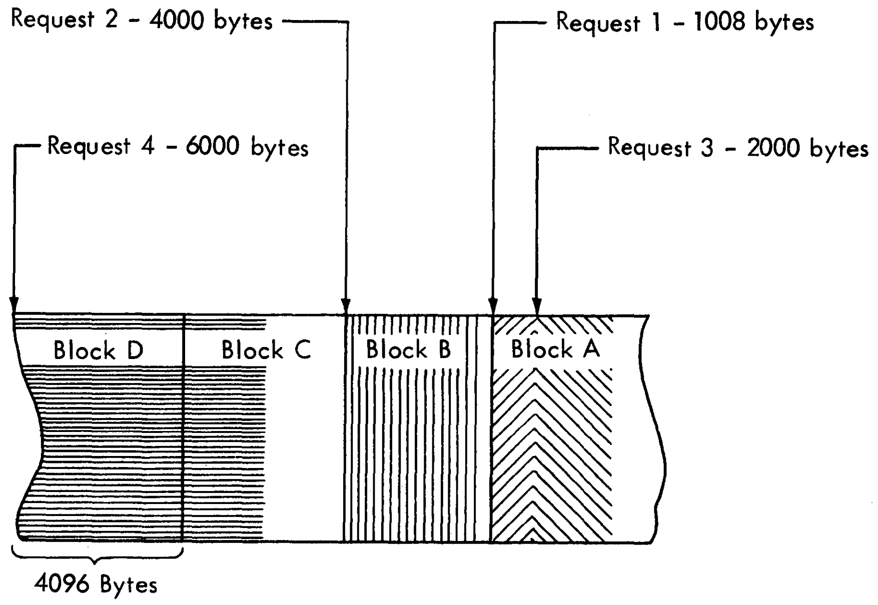


Figure 39. Virtual-storage control

program allocate the area from 2 contiguous blocks which were previously unassigned, Block D and Block C. These blocks are assigned to subpool 0.

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 then 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 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 operand can be written to request storage from subpools 0 to 127; if this operand is omitted, subpool 0 is assumed. The operands 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 operands 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 any of the operands described above is written in an ATTACH or a GETMAIN macro instruction, and that operand specifies a subpool which is not currently owned by or shared with the active task. If one of the ATTACH macro instruction operands 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 operand used.

Transferring Ownership: An owning task gives ownership of a subpool to a direct subtask by using the GSPV or GSPL operands 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 having 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.

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 and SHSPL operands 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 operand values should be contained in registers. For example, the POINT macro instruction requires that the DCB address and block address be coded as follows:

```
        [symbol]  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 one of the general registers 2-12 with the address, 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 operands that remain constant in every use of the macro instruction can be coded in the list form. These operands 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 operands previously designated. (Again, there are exceptions to this rule.)

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

Figure 40 shows the use of the list and execute forms of a DEQ macro instruction in a reenterable program. The length of the list constructed by the list form of the macro instruction is obtained by subtracting two symbolic addresses; virtual storage is allocated and the list is moved into the allocated area. The execute form of the DEQ macro instruction does not modify any of the operands 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 the routine labeled MOVERTN is valid for lengths up to 255 bytes only. Some macro instructions do produce lists greater than 255 bytes when many operands 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(1,4),0(3)
           ...
MACNAME    DEQ       (NAME1,NAME2,8,SYSTEM),RET=HAVE,MF=L
NSIADDR    ...       ...
NAME1      DC        CL8'MAJOR'
NAME2      DC        CL8'MINOR'
```

Figure 40. 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.

Freeing of Virtual Storage

As indicated previously, 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 using a DELETE macro instruction.
- If the load module was requested in a LINK, ATTACH, or XCTL macro instruction, that responsibility count is lowered 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.

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 operand was specified.

The date is returned in register 1 as packed decimal digits of the form 00yydddc, where yy are the last two digits of the year and ddd is the day of the year. C is the sign character hexadecimal F, which allows the year and day information to be unpacked directly for printing. One procedure used to request the date and time is shown in Figure 41.

The time of day is returned in register 0 in the form specified in the TIME macro instruction. The time of day is returned as an unsigned, 32-bit, binary number that specifies the elapsed number of either hundredths of a second, if BIN is coded, or timer units, if TU is coded. (A timer unit is equal to 26.04166 microseconds.) If DEC is coded or the operand is omitted, the time of day is returned as packed decimal digits of the form HHMMSSth (hours, minutes, seconds, tenths of a second, and hundredths of a second). The packed decimal digits can be unpacked by changing the "h" value to a zone sign and using an UNPK instruction or by inserting zones between each decimal digit. If MIC was specified, the time of day is returned in the doubleword supplied, with bit 51 the low-order digit of the value. Register 0 is set to 0, and register 15 has the return code for MIC.

All references to time of day and date 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 using the SET command with the DATE and CLOCK parameters. (For more information about the TOD clock, see IBM System/370 Principles of Operation.)

```
         ...
         TIME                      Request date and time
         ST    1,ANS               Store packed date
         UNPK  DOUBLE,ANS          Unpack date for printing
         OI    DOUBLE+7,X'F0'
         ...
ANS      DS    F                   Fullword for packed date
DOUBLE   DS    D                   Doubleword for unpacked date
```

Figure 41. Requesting the date and time

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. The time interval can be established by any one of the following five methods.

- BINTVL: Requires an unsigned 32-bit binary number, the low-order bit having a value of 0.01 seconds.

- TUINTVL: Requires an unsigned 32-bit binary number, the low-order bit having a value of 26.04166 microseconds (1 timer unit).

- DINTVL: Requires an 8-byte field containing unpacked decimal digits of the form HHMMSSth (hours, minutes, seconds, tenths and hundredths of a second, based on a 24-hour clock).

- TOD: Requires an 8-byte field similar to the field required for DINTVL. The control program interprets the time specified as the time of day at which the interval is to expire.

- MIC: Requires an 8-byte field containing an unsigned 64-bit binary number, bit position 51 of which is the low-order digit of the interval value.

When you test the time remaining in the interval with the TU option (default), the time remaining is returned as a 32-bit, unsigned, binary number in register 0, the low-order bit having a value of 26.04166 microseconds. If you test the time remaining with TTIMER MIC, the time remaining is returned in microseconds in the specified area. If the interval has already expired, the content of register 0 is set to 0.

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

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

```
         ...
         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 com-
                                          plete, repeat loop
         TTIMER   CANCEL                  If loop completes, cancel
                                          remaining time
         ...
NG       ...
         ...
         USING    FIXUP,15                Provide addressability
FIXUP    SAVE     (14,12)                 Save registers
         OI       TIMEXP,X'01'            Time interval expired,
                                          set switch in loop
         ...
         RETURN   (14,12)                 Restore registers
         ...
TIME     DC       X'00000200'             Timer is 5.12 seconds
TIMEXP   DC       X'00'                   Timer switch
```

Figure 42. Interval timing

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 OS users. A divide macro instruction (DXR) is provided for the models that have the extended-precision floating arithmetic facility and all eight instructions are provided for the models that do not. 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 _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 operand (the dividend) is divided by the second operand (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 _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 operand. The low-order 14 hexadecimal digits of the quotient fraction replace the high-order part of the first operand. The low-order 14 hexadecimal digits of the quotient fraction replace the low-order fraction of the first operand. 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 characterictic. 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 _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 then 0 and neither operand 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 operands 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 recorgnized. 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 43 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 43. 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 the 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 44 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 44. 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 44 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 44, 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 45.

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 46 shows the possible settings of the interruption code.

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 IEAXPDXR 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.

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

Figure 45. Return codes from the extended-precision floating-point simulator

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 stop 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 46. Interruption codes returned by the simulator

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. If your system has the MCS (multiple console support) option, 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. To use the single-line form, code the single-line message within apostrophes. The message that the operator receives does not contain these apostrophes. The message can include any character that is valid in a character (C-type) DC instruction, except the new-line control character (hexadecimal value 15). It is assembled as a variable-length record, which is written automatically; you do not have to provide a data control block.

To use the multiple-line form of the macro instruction, code the text of each line within apostrophes followed by a line type indicator. Enclose both of these items in one set of parentheses. Up to ten contiguous lines of information may be passed to the operator's console.

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

- Multiple-line WTO messages are not 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 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.
- 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 multiple-line form of the WTC macro instruction.

The message is routed (in a system with the MCS option) 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. For more information about routing codes, refer to Appendix C. (For RES users, the message is routed according to the user's queue identification number (QID).)

Disposition of the message (in a system with the MCS option) is indicated through the descriptor codes specified in the WTC 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. For more information about descriptor codes, refer to Appendix C.

A sample WTO macro instruction is shown in Figure 47. The routing code (ROUTCDE) and descriptor code (DESC) keyword parameters are ignored if the MCS option is not included in the system (except for WTP: see Writing to the Programmer below).

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

```
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 47. Writing to the operator

```
        ...
        XC     ECBAD,ECBAD        Clear ECB
        WTOR   'STANDARD OPERATING CONDITIONS?  REPLY YES OR NO',     C
               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 48. Writing to the operator with a reply

A sample WTOR macro instruction is shown in Figure 48. The reply is not necessarily available at the address you specified until a WAIT macro instruction has been issued.

When a WTOR macro instruction is issued with more than one routing code, any console within those areas 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. The master console operator 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. At system generation, your installation determines how many 176-byte SMBs (system message blocks) to allow. You can override this number at initial program load; however, the number of SMBs allowed must range from 1 to 20.

When you submit your job, you can specify the message output class for your messages by using the MSGCLASS parameter of the JOB statement. (For a description of the MSGCLASS parameter, refer to the Job Control Language Reference manual.) All WTO and WTOR messages within the number of SMBs allowed per job will appear in the designated message output class. When you exceed the number of allowable SMBs, no subsequent messages will appear in the message output class.

To write a message to the programmer, you must specify ROUTCDE=11 in the WTO or the WTOR macro instruction. If you use routing code 11 alone or together with other routing codes, the message goes to the message output class, as described above. The message can also go to the console(s) in the situations described by Figure 49.

If you specify a routing code of 11 (ROUTCDE=11)		
In this macro instruction:	In a system:	Your message goes to the:
WTO	With MCS	Message output class; Consoles designated to receive message with ROUTCDE=11
WTO	Without MCS	Message output class
WTOR	With MCS	Message output class; Master console
WTOR	Without MCS	Message output class; Master Console
If, in addition to routing code 11, you specify the appropriate routing code(s) in either a WTO or a WTOR macro instruction with or without MCS, the message appears on the console(s) designated to receive the routing code(s). In addition, the message appears in the same places as it does when you specify only routing code 11 (as shown above), with one exception. For WTOR with MCS, the message goes to the master console only if you specify that console's routing code.		

Figure 49. Using WTO and WTOR to write messages to the programmer

WRITING TO THE HARD-COPY LOG

When using an operating system that has the MCS (multiple console support) option, you can record information on the hard-copy log. Since the MCS option allows more than one console in a system, an installation might find it helpful to be able to record all the messages issued by and to a system. The hard-copy log provides a place to collect these messages and therefore allows an installation to review system activity by reviewing message activity.

Since the hard-copy log is optional, you should know whether your system was generated with it. The hard-copy log is either an operator's console with output capability or the system log.

To record information on the hard-copy log, you use the WTO or WTOR macro instruction. Your installation must have decided which system functions are to be logged and assigned appropriate routing codes to the hard-copy log. The routing codes that you assign to your WTO or WTOR macro instruction are compared to the routing codes assigned to the log. If one or more codes match, the message is entered in the log. This means you do not have to issue a WTL macro instruction to record system and problem program information when the same information is going to the operator. You must, however, know which system functions the log is recording and assign an appropriate routing code to your WTO or WTOR macro instruction.

For each entry in the hard-copy log, both the time when the message is received by the system and the routing codes for the message are appended to the beginning of the message text. Recording the time that the message was received, a procedure called time stamping, allows you to obtain a chronological record of system activity. For a system that does not have the timer option, the space for time stamping is filled with zeros.

Whether the hard-copy log is the operator's console or the system log, the hard-copy log information cannot be confused with other information. This is because the hard-copy log entries are prefixed with the time stamp and the routing codes.

WRITING TO THE SYSTEM LOG

The system log (optional in VS1, standard in VS2) consists of two SYSOUT data sets 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" operand of the WTL macro instruction.

The data set receiving data from the control program, user programs, and operators is the primary data set. The data set being written, or waiting to be written, to a system output device is the alternate data set. The primary data set, the one that is open and receiving input, is logically connected to two buffers. The control program fills one buffer and writes it to the primary data set while filling the other buffer. The alternate data set has been logically disconnected from the buffers because it has been filled and must wait to be written to a system output device. After being written to a system output device, the alternate data set can be used again to receive input. When receiving input, the alternate data set becomes the primary data set.

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 primary data set. The control program writes the text of your WTL macro instruction on the master console instead of on the system log if one of the following two conditions exists:

- The system log is not supported.
- The system log is supported, but the system log data sets are temporarily inactive because both are full and waiting to be written.

Your installation probably has an operator procedure to follow for both of the above conditions.

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.

GENERALIZED TRACE FACILITY INTERFACE

One of the capabilities of GTF (generalized trace facility) is the recording of data originated by application programs. The interface between the application programs and GTF is the GTRACE macro instruction. (For a complete discussion of GTF, see the Service Aids publication.)

GTRACE allows from 1 to 256 bytes of data to be entered in a GTF buffer and recorded. When the GTRACE macro instruction is executed, GTF must be active and conditioned to receive application data and to record this data on an external device; otherwise the data will not be accepted. Return codes are used to indicate the result of the operation.

Recorded data is processed by the edit function of the HMDPRDMP service aid. If you want more than a hexadecimal dump of the records, you may prepare formatting routines for use with the HMDPRDMP edit function. Association between your recorded data and the formatting routine that is to process it is established by entering a format identifier in the GTRACE macro instruction. This identifier defines the formatting routine that is to process the record. For a more complete discussion of HMDPRDMP, see the Service Aids publication.

To use the GTRACE macro instruction, specify the address and the number of bytes of data to be entered, along with an event identifier. A unique event identifier may be specified each time the GTRACE macro instruction is used. This identifier may be used, for example, in output record identification. The optional FID parameter indicates the formatting routine to be used by HMDPRDMP in processing the record. In Figure 50, 200 bytes of data, beginning at location AREA, are to be recorded with an event identifier of 37. In Figure 50 HMDUSR28 is designated.

```
GTRACE  DATA=AREA,LNG=200,ID=37,FID=40
```

Figure 50. Using the GTRACE macro instruction

CHAPTER 9: 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 operands, and a control program list, resulting from the remaining operands. 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 operand 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 form.

CODING AIDS

The symbols [ ], { }, ,..., and ____ are used to indicate how a macro instruction may be written. DO NOT CODE THESE SYMBOLS. The specific meanings of these symbols are given at the bottom of each page on which they are used; their general definitions are given below:

[ ] indicates optional operands. The operand enclosed in the brackets (for example, [VL]) may or may not be coded, depending on whether or not the associated option is desired. If more than one item is enclosed in brackets (for example, $\begin{bmatrix}\text{REREAD}\\\text{LEAVE}\end{bmatrix}$), one or none of the items may be coded.

{ } indicates that a choice must be made. One of the operands from the vertical stack within braces (for example, $\left\{\begin{bmatrix}\text{input}\\\text{output}\end{bmatrix}\right\}$) must be coded, depending on which of the associated services is desired.

,... indicates that more than one set of operands may be designated in the same macro instruction.

____ indicates a value that is used in default of a specified value. This value is assumed if the operand is not coded.

WRITING THE MACRO INSTRUCTIONS

The system macro instructions are written in the assembler language, and are subject to the rules contained in the publication VS Assembler Language. Write system macro instructions, like all assembler language instructions, in the following format:

Name	Operation	Operands	Comments
symbol or blank	Macro name	Blank, or one or more operands separated by commas	

Use the operands to specify the services and options to be performed; write them according to the following general rules:

- If the selected operand is written in all capital letters (for example, STEP, DUMP, RET=USE), code the operand exactly as shown.

- If the selected operand is written in lower case letters, substitute the indicated value, address, or name.

- If the selected operand is a combination of capital and lower case letters separated by an equal sign (for example, EP=entry point name), code the capital letters and equal sign as shown, then make the indicated substitution.

- Code commas and parentheses exactly as shown, only omit the comma following the last operand coded. The use of commas and parentheses is indicated by brackets and braces, exactly as operands.

When substitution is required, the method of specifying the operand depends on the requirements of the control program. Descriptions of list and execute form macro instructions indicate specifically how the operands should be coded; the descriptions of the standard forms of the macro instructions indicate only what is to be coded. Appendix A shows specifically how the operands are to be coded. The table in Appendix A contains all of the operands for which substitution is required and indicates the allowable ways of writing the operands. The classifications are as follows:

SYM
: is any symbol valid in the assembler language.

DEC DIG
: is any decimal digit up to the value indicated in the associated macro instruction description. If both SYM and DEC DIG are checked, an absolute expression is also allowed.

REGISTER
: is always coded within parentheses, as follows:

(2-12)	- one of general registers 2 through 12 that you have previously loaded with the right-adjusted value or address specified in the associated macro instruction description. The unused high-order bits must be set to zero. You may designate the register symbolically or with an absolute expression.
(1)	- general register 1, previously loaded as indicated above. Designate the register as (1) only.
(0)	- general register 0, previously loaded as indicated above. Designate the register as (0) only.

RX TYPE
: is any address that is valid in an RX-type instruction (for example, LA).

ADCON TYPE
: is any address that may be written in an A-type address constant.

CONTINUATION LINES

You can continue the operand 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 operand coding) in column 72 of the line.

2. Continue the operand field on the next line, starting in column 16. All columns to the left of column 16 must be blank.

You can code the operand field being continued in one of two ways. Code the operand field through column 71, with no blanks, and continue in column 16 of the next line; or truncate the operand 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 51 shows an example of each method. Additional information on the continuation of any assembler language macro instruction is provided in the publication VS Assembler Langauge.

ADDITIONAL MACRO INSTRUCTIONS

Descriptions and definitions of the following macro instructions are contained in the publications VS1 Planning and Use Guide and VS2 Planning and Use Guide.

ATLAS	PURGE
CATALOG	RESERVE
CIRB	RESTORE
CVT	SMFWTM
EOV	STAE
EXTRACT	SYNCH
MODESET	TESTAUTH

Name	Operation	Operand	Comments		
NAME1	OP1	OPERAND1,OPERAND2,OPERAND3,OPERAND4,OPERA		X	
		ND5,OPERAND6	THIS IS ONE WAY		
NAME2	OP2	OPERAND1,OPERAND2,	THIS IS	X	
		OPERAND3,	ANOTHER WAY	X	
		OPERAND4			

Figure 51. Continuation coding

CHAPTER 10: DESCRIPTIONS OF THE MACRO INSTRUCTIONS

The macro instructions are described in alphabetical order. For your convenience, the upper right-hand corner of each page contains the name of the macro instruction described on the page.

ABEND -- Abnormally Terminate a Task

Use the ABEND macro instruction to abnormally terminate the active task and all its subtasks. ABEND can request a dump of all 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. 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 action is 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 operand description.
- One end-of-task exit routine is selected to be given control; the one specified in the ATTACH macro instruction that created the task that was active when ABEND was issued. The exit routine is given control when the <u>originating</u> 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:

[symbol]	ABEND	completion code[,DUMP][,STEP]

completion code
: is a maximum of 4095. The value may be specified symbolically, as a decimal digit, or as one of the registers 1 through 12 (in parentheses). Using a value greater than 4095 causes unpredictable user or system completion codes, or both. If the job step is to be terminated, the completion code is recorded as user code on the system output device. If the job step is not to be terminated, the completion code is placed in the task control block of the active task and in the event control block specified in the ECB operand of the ATTACH macro issued to create the active task.

DUMP
: is written as shown. It is used to request a dump of all virtual storage areas assigned to the task and all control blocks pertaining to the task. A sample abnormal termination dump is contained in the VS1 and VS2 <u>Debugging Guides</u>.

A separate dump is provided for each of the tasks being terminated as a result of ABEND. In addition, a dump of the control blocks and save areas is provided for each of the higher level tasks that are direct predecessors of the task being terminated. You should provide a //SYSABEND or a //SYSUDUMP DD statement; if you do not, you receive only an indicative dump. If the operand is omitted or if insufficient storage is available in the partition for the abnormal termination to be performed, no dump is provided.

[ ] indicates optional name or operands.

STEP
is written as shown. It indicates that the entire job step of the active task is to be abnormally terminated.

Note: During the ABEND process, the terminating task is multiprogrammed with other tasks in the system. Therefore, the resulting storage dump may reflect changes that occur during ABEND execution to storage areas not uniquely related to the abnormally terminating task.

ATTACH -- Create a New Task (VS1)

Use the ATTACH macro instruction to create a new task. The new task is a subtask of the originating task. Both tasks exist in the same partition and compete for execution with each other and with all other tasks in the system, based on dispatching priority. The limit and dispatching priorities of the new subtask are the same as those of the originating task unless modified in the ATTACH macro instruction.

The originating task specifies the entry point of the program to be given control when the subtask begins execution. The specified entry point must be a member name or an alias in a directory of a partitioned data set, or it must have been specified in an IDENTIFY macro instruction. If the specified entry point cannot be located, the new subtask is abnormally terminated. The load module containing the program is brought into storage if a usable copy is not available.

A linkage relationship is established between the originating task and the subtask by the task control block. The originating task can pass a parameter list to the subtask; it can also provide an event control block in which termination of the new task is posted, and an exit routine to be given control upon subtask termination. The address of the task control block for the subtask is returned in register 1.

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

For systems with the time-slicing option, the dispatching priority determines whether the new task will participate in time-slicing. The new task executes asynchronously to the calling task, but within the same partition.

Write the standard form of the ATTACH macro as follows. Information about the list and execute forms follows this description.

[symbol]	ATTACH	{EP=symbol
		EPLOC=address of name
		DE=address of list entry}
		[,DCB=dcb address]
		[,PARAM=(addresses)[,VL=1]]
		[,ECB=ecb address][,ETXR=exit routine address]
		[,LPMOD=number][,DPMOD=number]

EP=
: is the entry name in the load module to be given control.

EPLOC=
: is the virtual storage address of the entry name. Pad the name with blanks to eight bytes, if necessary.

DE=
: is the address of the name field of a 58-byte list entry for the entry name. The list entry is constructed using the BLDL macro instruction. The DCB operand must indicate the same data control block used in the BLDL macro instruction.

[ ] indicates optional name or operand; select one from vertical stack with { }.

DCB=
is the address of the data control block for the partitioned data set containing the entry name. Specifying an address of zero or omitting the DCB operand indicates the data set is in the link library or the job step library.

PARAM=
is one or more address parameters, separated by commas, 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 operand is omitted, register 1 is not altered.

VL=1
is written as shown. Use it only if PARAM is designated, and 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.

ECB=
is the address of an event control block (fullword on a fullword boundary) 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 you code this operand, you must also issue a DETACH macro instruction to remove the subtask from the system after the subtask has been terminated.

ETXR=
is 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 you code this operand, you must also issue a DETACH macro instruction 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. It operates logically as a subroutine and must return control to the control program.

LPMOD=
is the number to be subtracted from the current limit priority of the originating task. The result is the limit priority of the new task. If omitted, the current limit priority of the originating task is assigned as the limit priority of the new task.

DPMOD=

is the signed number 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 you specify a register, a negative number must be in two's complement form in the register. If you omit this operand, the dispatching priority assigned is the smaller of either the new task's limit priority or the originating task's dispatching priority.

ATTACH -- Create a New Task (VS2)

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 dispatching priority determines whether or not the new task participates in time slicing or in the APG (automatic priority group).

The load module containing the program to be given control is brought into virtual storage if a usable copy is not available in virtual storage. The issuing program can provide an event control block, in which termination of the new task is posted, an exit routine to be given control when the new task is terminated, and a parameter list whose address is passed in register 1 to the new task. If the ECB or ETXR operands are coded, a DETACH macro instruction must be issued to remove the subtask from the system before the program that issued the ATTACH macro instruction terminates. If the ECB or ETXR operands are 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 storage 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.

For further discussions of time slicing and the use of an existing copy of a load module, refer to Part I.

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

```
┌───────────┬─────────┬──────────────────────────────────────────────────────────┐
│ [symbol]  │ ATTACH  │ (EP=symbol              )   [,DCB=dcb address]           │
│           │         │ <EPLOC=address of name  >                                │
│           │         │ (DE=address of list entry)                               │
│           │         │                                                          │
│           │         │ [,LPMOD=number]             [,DPMOD=number]              │
│           │         │ [,PARAM=(addresses)[,VL=1]]                              │
│           │         │ [,ECB=ecb address][,ETXR=exit routine address]           │
│           │         │                                                          │
│           │         │ [,GSPV=number          ]  [,SHSPV=number          ]      │
│           │         │ [,GSPL=address of list ]  [,SHSPL=address of list ]      │
│           │         │                                                          │
│           │         │           {YES}                                          │
│           │         │ [,SZERO={NO }]                                           │
│           │         │                                                          │
│           │         │ [,TASKLIB=dcb address]                                   │
│           │         │                                                          │
│           │         │ [,STAI=(exit address[,parameter list address])]          │
│           │         │                                                          │
│           │         │          (NONE    )                                      │
│           │         │          <HALT    >               {YES}                  │
│           │         │ [,PURGE=(QUIESCE )] [,ASYNCH={NO }]                      │
└───────────┴─────────┴──────────────────────────────────────────────────────────┘
```

EP=
  is the entry name in the load module to be given control.

EPLOC=
  is the virtual storage address of the entry name. The name must be padded with blanks to eight bytes, if necessary.

DE=
  is the address of the name field of a list entry for the entry name. The list entry is constructed using the BLDL macro instruction. The DCB operand must indicate the same data control block used in the BLDL macro instruction.

DCB=
  is the address of the data control block for the partitioned data set containing the entry name. The address of the data control block for either the link or job library is designated by specifying an address of 0 or by omitting the DCB operand.

LPMOD=
  is the number to be subtracted from the current limit priority of the originating task. The result is the limit priority of the new task. If omitted, the current limit priority of the originating task is assigned as the limit priority of the new task.

DPMOD=
  is the signed number 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.

[ ] indicates optional name or operand; select one from vertical stack within { }; select one or none from vertical stack within [ ].

If a register is designated, a negative number must be in two's complement form in the register. If this operand 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=
is one or more address parameters, separated by commas, to be passed to the called program. Each address is expanded in line 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 operand is omitted, register 1 is not altered.

VL=1
is written as shown. It can be designated only if PARAM is designated, and should be used 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.

ECB=
is 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 operand is coded, a DETACH macro instruction must be issued to remove the subtask from the system after the subtask has been terminated.

ETXR=
is 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 operand 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=
is a virtual storage subpool number. Ownership of the specified virtual storage subpool is assigned to the new task. Programs of the originating task can no longer use the associated virtual storage area.

GSPL=
: is the address of a list of virtual storage subpool numbers. 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. Ownership of each of the specified virtual storage subpools is assigned to the new task. Programs of the originating task can no longer use the associated virtual storage areas.

SHSPV=
: is a virtual storage subpool number. Programs of both the originating task and the new task can use the associated virtual storage area.

SHSPL=
: is the address of a list of virtual storage subpool numbers. 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. Programs of both the originating task and the new task can use the associated virtual storage areas.

SZERO=
: is used to indicate 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. YES is assumed if this operand is omitted.

TASKLIB=
: is used to indicate whether a task library DCB address has been supplied. If an address is supplied, it is stored in TCBJLB. Otherwise, TCBJLB is propagated from the originating task.

STAI=
: is used to indicate whether a STAI SCB is to be created; any STAI SCBs queued to the originating task are propagated to the new task. The first address supplied should be the address of the STAI exit routine which is to receive control if the subtask abnormally terminates. The STAI exit routine must be in virtual storage at the time of abnormal termination. The second address is the address of a parameter list which may be used by the STAI exit routine. A BC (basic control) mode PSW is reflected in the STAE work area.

PURGE=
: is used to indicate what action is to be taken with regard to I/O operations when the subtask is abnormally terminated. This operand is used only in conjunction with the STAI= operand. 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). The meaning of the PURGE= operand is exactly as specified in the STAE macro description. If omitted, QUIESCE is assumed.

ASYNCH=
: is used to indicate whether asynchronous exits are to be allowed when a subtask ABEND occurs. This operand is used only in conjunction with the STAI= operand. The meaning of the ASYNCH= operand is exactly as specified in the STAE macro description. If omitted, NO is assumed.

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 description of the standard form of ATTACH explains the function of each operand. The description of the standard form also indicates which operands are always optional and which are required in at least one of the pair of list and execute forms. The operand combinations in the shaded area of the format description may only be used in a VS2 system. The TQE and FPREGSA operands are only used in a VS1 system. The format description below indicates the optional and required operands in the list form only.

```
[symbol]   ATTACH   (EP=symbol                   )
                    {EPLOC=address of name        }
                    (DE=address of list entry    )
                    [,DCB=dcb address][,ECB=ecb address]
                    [,ETXR=exit routine address][,LPMOD=number]
                    [,DPMOD=number],SF=L
                            {YES}              {YES}
                    [,TQE={NO }][,FPREGSA={NO }]   VS1 only
                    [,GSPV=number][,GSPL=address of list]
                    [,SHSPV=number][,SHSPL=address of list]
                               {YES}
                    [,SZERO={NO }][,TASKLIB=dcb address]
                    [,STAI=(exit address[,parameter list address])]
                              {NONE   }           {YES}
                     ,PURGE={HALT   }  [,ASYNCH={NO }]
                              {QUIESCE}
```

symbol
    is any symbol valid in the assembler language.

address
    is any address that may be written in an A-type address constant.

number
    is any absolute expression valid in the assembler language.

SF=L
    indicates the list form of ATTACH.

[ ] indicates optional name or operands; select one from vertical stack within { }.

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, operands that require use of the control program parameter list cause that list to be constructed inline as part of the macro expansion.

The description of the standard form of ATTACH explains the function of each operand. The description of the standard form also indicates which operands are always optional and which are required in at least one of the pair of list and execute forms. The operand combinations in the shaded area of the format descriptions may only be used in a VS2 system. The TQE and FPREGSA operands are only used in a VS1 system. If specified in the list form, you need not respecify them in the execute form. The format description below indicates the optional and required operands in the execute form only.

```
┌───────────┬──────────┬─────────────────────────────────────────────────┐
│ [symbol]  │ ATTACH   │ (EP=symbol              )                       │
│           │          │ {EPLOC=address of name  }                       │
│           │          │ (DE=address of list entry)                      │
│           │          │ [,DCB=dcb address]                              │
│           │          │ [,PARAM=(addresses)[,VL=1]]                     │
│           │          │ [,ECB=ecb address][,ETXR=exit routine address]  │
│           │          │ [,LPMOD=number][,DPMOD=number]                  │
│           │          │       {YES}             {YES}                   │
│           │          │ [,TQE={NO }][,FPREGSA={NO }]   VS1 only         │
│           │          │ (,MF=(E,{problem program list address})  )      │
│           │          │ (       {          (1)               }   )      │
│           │          │ {,SF=(E,{control program list address})  }      │
│           │          │ (       {          (15)              }   )      │
│           │          │ (,MF=(E,{address}),SF=(E,{address})      )      │
│           │          │ (       {  (1)  }        {  (15) }       )      │
│           │          │ [,GSPV=number][,GSPL=address of list]           │
│           │          │ [,SHSPV=number][,SHSPL=address of list]         │
│           │          │          {YES}                                  │
│           │          │ [,SZERO={NO }][,TASKLIB=dcb address]            │
│           │          │ [,STAI=(exit address[,parameter list address])] │
│           │          │          {NONE   }                  {YES}       │
│           │          │ [,PURGE= {HALT   }]         [,ASYNCH={NO }]     │
│           │          │          {QUIESCE}                              │
└───────────┴──────────┴─────────────────────────────────────────────────┘
```

symbol
    is any symbol valid in the assembler language.

address
    is any address that is valid in an RX-type instruction, or one of general registers 2 through 12, previously loaded with the indicated address. You may specify the register symbolically or with an absolute expression; always code it within parentheses.

number
    is any absolute expression that is valid in the assembler language, or one of general registers 2 through 12, previously loaded with the indicated value. You may specify the register symbolically or with an absolute expression; always code it within parentheses.

[ ] indicates optional name or operand; select one from vertical stack within { }.

```
MF=(E,{problem program list address})
      {          (1)                }
```

indicates the execute form of the macro instruction using a remote problem program parameter list. Any control program parameters specified are provided in a control program parameter list expanded inline. If you also specify PARAM, the address parameters are placed on contiguous fullword boundaries, beginning at the address specified in the MF operand and sequentially overlaying corresponding fullwords in the existing list. The address of the problem program parameter list can be coded as described under "address," or can be loaded into register 1, in which case MF=(E,(1)) should be coded.

```
SF=(E,{control program list address})
      {          (15)               }
```

indicates the execute form of the macro instruction using a remote control program parameter list. Any problem program parameters specified are provided in a problem program parameter list expanded in line. The address of the control program parameter list can be coded as described under "address," or can be loaded into register 15, in which case SF=(E,(15)) should be coded.

```
MF=(E,{address}),SF=(E,{address})
      { (1)   }        { (15)  }
```

indicates the execute form of the macro instruction using both a remote problem program parameter list and a remote control program parameter list. The addresses of the parameter lists are coded or loaded into registers 1 and 15, as explained above.

CALL -- Pass Control to a Control Section

CALL 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. (15) must not be designated in an exclusive call. 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. If (15) is designated, the load module containing the entry point must be in virtual storage, and register 15 must contain the address of the 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. Information about the list and execute forms follows this description.

[symbol]	CALL	{entry point name / (15)} [,(address parameters)[,VL]] [,ID=number]

entry point name
: is the name of the entry point to be given control; the name is used in the macro instruction as the operand of a V-type address constant; before execution it must be resolved to a virtual address by linkage editing. If (15) is designated, register 15 must contain the address of the entry point to be given control.

address parameters
: are one or more address parameters, separated by commas, to be passed to the called program. Each address is expanded, in the order designated, to a fullword on a fullword boundary. When control is passed, register 1 contains the address of the first parameter. If no address parameters are designated, the contents of register 1 are not changed.

VL
: is written as shown. It can be designated only if address parameters are designated. Use it only when a variable number of parameters can be passed to the called program. VL causes the high-order bit of the last address parameter in the macro expansion to be set to 1; the bit can be checked by the called program to find the end of the list.

[ ] indicates optional name or operand; select one from vertical stack within { }.

ID=

maximum value is $2^{16}-1$. The last fullword of the macro expansion is a NOP instruction containing the ID value in the low-order two bytes. When the called program is given control, the address resides at 2 bytes past the location pointed to by register 14.

Upon entry to the called program, register contents are as follows:

Register	Contents
1	Address of parameter list if present.
14	Return address.
15	Entry point 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 description of the standard form of the CALL macro instruction explains the function of each operand. The description of the standard form also indicates which operands are always optional and which are required in at least one of a pair of list and execute forms. The format description below indicates the optional and required operands in the list form only. The comma before the parenthesis must be coded to indicate the absence of the entry name operand, which is not allowed in the list form.

[symbol]	CALL	,(address parameters)[,VL],MF=L

symbol
    is any symbol valid in the assembler language.

address
    is any address that may be written in an A-type address constant.

MF=L
    indicates the list form of the CALL macro instruction.

[ ] indicates optional name or operands.

CALL -- Execute Form

A remote problem program parameter list is referred to and can be modified by the execute form of the CAIL macro instruction. Only executable instructions and a VCON of the entry point are generated.

The description of the standard form of the CALL macro instruction explains the function of each operand. The description of the standard form also indicates which operands are always optional and which are required in at least one of the pair of list and execute forms. The format description below indicates the optional and required operands in the execute form only.

```
┌──────────┬──────┬─────────────────────────────────────────────────────┐
│ [symbol] │ CALL │ {entry point name}[,(address parameters)][,VL]      │
│          │      │ {      (15)      }                                  │
│          │      │ [,ID=number],MF=(E,{problem program list address}) │
│          │      │                    {           (1)              }  │
└──────────┴──────┴─────────────────────────────────────────────────────┘
```

name, symbol
  is any symbol valid in the assembler language.

address
  is any address of an existing address constant that is valid in an RX-type instruction, or one of general registers 2 through 12, previously loaded with the indicated address. The register may be designated symbolically or with an absolute expression, and is always coded within parentheses.

number
  is any absolute expression that is valid in the assembler language, or one of general registers 2 through 12, previously loaded with the indicated value. The register may be designated symbolically or with an absolute expression, and is always coded within parentheses.

```
MF=(E,{problem program list address})
      {           (1)              }
```
  indicates the execute form of the macro instruction using a remote problem program parameter list. The address of the problem program parameter list can be coded as described under "address," or can be loaded into register 1, in which case code MF=(E,(1)). Register 1 or the problem program list address must point to the list form of CALL. If the address parameter is also specified, the ADCONs of the parameter are placed on contiguous fullword boundaries beginning at the address specified in the MF operand, and sequentially overlaying corresponding fullwords in the existing list.

[ ] indicates optional name or operand; select one from vertical stack within { }.

CHAP -- Change Dispatching Priority

CHAP changes the dispatching priority of the task or any of its subtasks. CHAP may also change the limit priority of a subtask. (See the section "Priority of Subtasks" in this publication. The algebraic sum of the priority change 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 change 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 change 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.

Notes: The limit priority of the job step task depends on the PRTY parameter of the JOB statement. The dispatching priority of any task determines whether or not the task participates in time slicing (applicable when the operating system includes the time-slicing option). For more details, refer to Time Slicing in the Services section.

If the CHAP issuer is in the dynamic dispatching group, the CHAP request is ignored. If the CHAP issuer is not in the dynamic dispatching group, and the resultant dispatching priority for the CHAPed task is equal to the dispatching priority of the dynamic dispatching group, the CHAPed TCB is moved in the ready queue to the position immediately in front of the dynamic dispatching group. You may not CHAP into, or out of a dynamic dispatching group.

The standard form of CHAP is written as follows:

[symbol]	CHAP	priority change value [,tcb location address / ,'S']

priority change value
: is 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 should be in two's complement form.

tcb location address
: 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. The address of the task control block is an output of the ATTACH macro. 'S' indicates that the priority of the active task is to be changed. 'S' is assumed if the operand is omitted or if it is coded to specify a zero address.

[ ] indicates optional name or operand; select one or none from vertical stack within [ ]; ____ indicates an assumed value.

DELETE -- Relinquish Control of a Load Module

DELETE 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 name specified in DELETE must be the same as that specified in the LOAD macro instruction that brought the load module into storage; it must be issued by the same task that issued the LOAD macro instruction.

The DELETE macro is written as follows:

```
┌───────────┬─────────┬────────────────────────────────┐
│ [symbol]  │ DELETE  │ ⎧EP=symbol                    ⎫ │
│           │         │ ⎨EPLOC=address of name        ⎬ │
│           │         │ ⎩DE=address of list entry     ⎭ │
└───────────┴─────────┴────────────────────────────────┘
```

EP=
: is the entry name used in the associated LOAD macro instruction.

EPLOC=
: is the address of the entry name described above. Pad the name with blanks to eight bytes, if necessary.

DE=
: is the address of the name field of a 58-byte list entry for the entry name described above, constructed using the BLDL macro instruction.

When control is returned, register 15 contains a 0 if the operation was completed successfully. Register 15 contains a 4 if a LOAD was not issued for the task issuing the DELETE instruction or if the responsibility count had previously become zero.

[ ] indicates optional name; select one from vertical stack within { }.

DEQ -- Release a Serially Reusable Resource

DEQ removes control of one or more (maximum is 255) serially reusable resources from the active task. It can also be used to determine whether control of the resource is currently assigned to or requested for 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 an invalid address of a resource results in abnormal termination of the task.

The standard form of the DEQ macro instruction is written as follows. Information about the list and execute forms follows this description.

```
┌─────────┬───┬────────────────────────────────────────────────────────┐
│[symbol] │DEQ│(qname address,rname address,[rname length],┌STEP  ┐,...)│
│         │   │                                            └SYSTEM┘     │
│         │   │[,RET=HAVE]                                              │
└─────────┴───┴────────────────────────────────────────────────────────┘
```

qname address
: is the address in virtual storage of an eight-character name (padded with blanks on the right if necessary). Every program issuing a request for a serially reusable resource must use the same qname and rname to represent the resource. The name should not start with SYS, so that it will not conflict with system names. The name must be the same qname previously specified for the resource in an ENQ macro instruction.

rname address
: is the address in virtual storage of the name used in conjunction with the qname 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.

rname length
: is the length of the rname described above. The length must have the same value as specified in the previous ENQ macro instruction. If the operand 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 a value of 0. If you specify 0, the length of the rname must be contained in the first byte at the rname address designated above.

STEP or SYSTEM
: is written as shown. You must specify the same STEP or SYSTEM option as you used in the ENQ macro instruction requesting the resource. You can specify up to 255 resources.

RET=HAVE
: is written as shown. It 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. If the operand is omitted, the request for release is unconditional, and the active

----

[ ] indicates optional name or operand; select one or none from vertical stack within [ ]. ,... indicates that more than one resource can be specified in the DEQ macro instruction, to a maximum of 255.

task is abnormally terminated if it has been assigned control of the resources. The results of conditional requests are indicated by the return codes shown in Figure 52.

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 53. 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 52.

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 52. DEQ macro instruction return codes

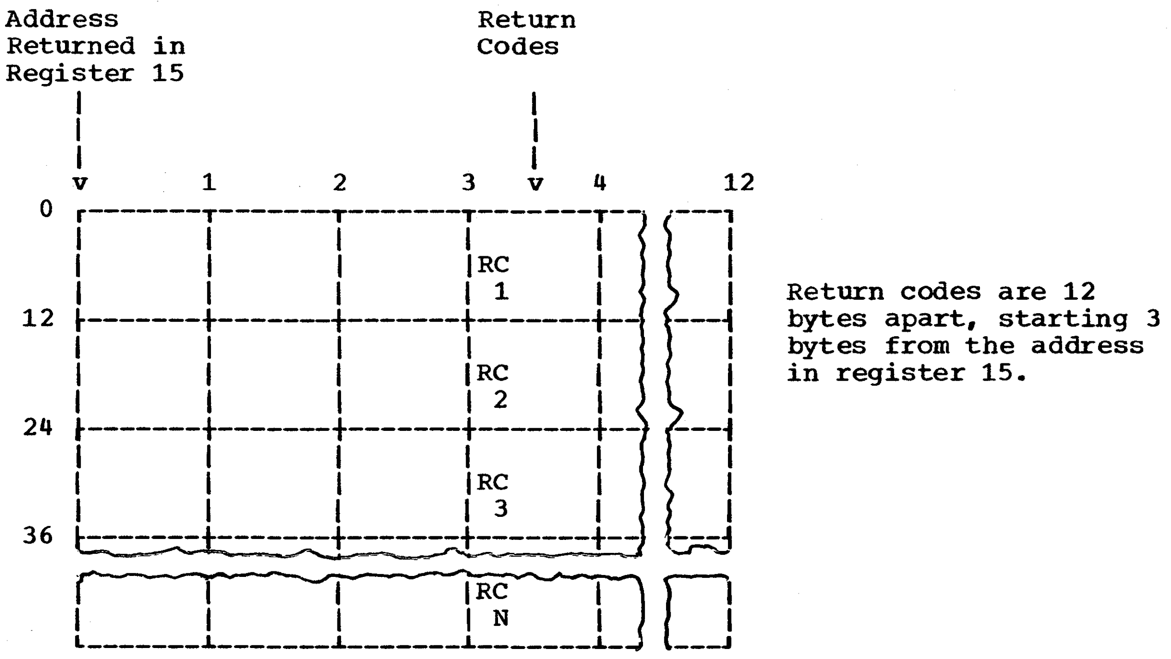

Address
Returned in
Register 15
Return
Codes
v
1
2
3
v
4
12
0
12
24
36
RC 1
RC 2
RC 3
RC N
Return codes are 12 bytes apart, starting 3 bytes from the address in register 15.

Figure 53. Return code area used by DEQ

DEQ -- List Form

Use the list form of DEQ to construct a control program parameter list. You can specify any number of resources in DEQ; therefore, the number of qname and rname combinations in the list form of DEQ must be equal to the maximum number of qname and rname combinations in any execute form of DEQ that refers to that list form.

The description of the standard form of the DEQ macro instruction explains the function of each operand. The description of the standard form also indicates which operands are always optional and which are required in at least one of the pair of list and execute forms. The format description below indicates the optional and required operands in the list form only.

```
┌───────────┬──────┬──────────────────────────────────────────────────────┐
│ [symbol]  │ DEQ  │ ([qname address],[rname address],[rname length],     │
│           │      │ ⎡SYSTEM⎤,...)[,RET=HAVE],MF=L                        │
│           │      │ ⎣STEP  ⎦                                             │
└───────────┴──────┴──────────────────────────────────────────────────────┘
```

symbol
    is any symbol valid in the assembler language.

address
    is any address that may be written in an A-type address constant.

length
    is any absolute expression valid in the assembler language.

MF=L
    indicates the list form of the DEQ macro instruction.

[ ] indicates optional name or operand; ,... indicates that more than one qname and rname combination with associated options can be coded. Select one or none from vertical stack within [ ].

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 description of the standard form of DEQ explains the function of each operand. The description of the standard form also indicates which operands are always optional and which operands are required in at least one of the pair of list and execute forms. The format description below indicates the optional and required operands in the execute form only.

```
┌───────────┬─────┬──────────────────────────────────────────────────┐
│ [symbol]  │ DEQ │ [([qname address],[rname address],[rname length], │
│           │     │ ┌SYSTEM┐,...)] ┌,RET=HAVE┐                        │
│           │     │ └STEP  ┘       └,RET=NONE┘                        │
│           │     │ ,MF=(E,{control program list address})            │
│           │     │        {         (1)                 }            │
└───────────┴─────┴──────────────────────────────────────────────────┘
```

symbol
    is any symbol valid in the assembler language.

address
    is any address that is valid in an RX-type instruction, or one of general registers 2 through 12, previously loaded with the indicated address. You may designate the register symbolically or with an absolute expression; always code it within parentheses.

length
    is any absolute expression that is valid in the assembler language, or one of general registers 2 through 12, previously loaded with the indicated value. You may designate the register symbolically or with an absolute expression; always code it within parentheses.

RET=NONE
    specifies an unconditional request to release all of the resources. The request is processed as though no RET operand had been coded.

```
MF=(E,{control program list address})
      {          (1)                }
```
    indicates the execute form of the macro instruction using a remote control program parameter list. Code the address of the control program parameter list as described under "address," or you can load the address into register 1, in which case you should code MF=(E,(1)).

[ ] indicates optional name or operand; select one from vertical stack within { }; ,... indicates that more than one qname and rname combination with associated options can be coded. Select one or none from vertical stack within [ ].

DETACH -- Delete a Subtask (VS1)

DETACH removes a previously terminated subtask from the system by releasing the storage containing the TCB (task control block) of the subtask. The subtask must be detached by the task that created it or by a task using the same TCB as the creating task, and the ATTACH macro instruction used to create the subtask must have specified the ECB or ETXR operand. You can issue a DETACH only for subtasks created by an active task.

All subtasks created through execution of the ATTACH macro instruction specifying ECB or ETXR must be detached before the originating task terminates. Failure to detach the subtasks results in abnormal termination of the originating task when it attempts normal termination. If neither ECB nor ETXR is specified in the ATTACH macro instruction, DETACH should not be issued.

If a DETACH is issued for a subtask that has not completed execution, that subtask and all its subtasks are abnormally terminated. If an ECB for recording the subtask's termination is present, it is posted, but an end-of-task exit routine is ignored. Instead, the detaching task regains control at the next sequential instruction.

If an invalid TCB address is passed to DETACH, the task that issued the DETACH is abnormally terminated.

If a subtask was created by an ATTACH macro instruction that specified the ECB operand, and the subtask has not yet terminated, DETACH abnormally terminates the subtask, specifies the completion code, and posts it in the ECB.

If a subtask was created by an ATTACH macro instruction that did not specify ECB or ETXR, and the task has terminated, an attempt to detach the subtask results in abnormal termination of the task issuing DETACH.

The DETACH macro instruction is written as follows:

[symbol]	DETACH	tcb location address

tcb location address
: is the virtual storage address of a fullword on a fullword boundary. The fullword contains the address of the task control block for the subtask to be removed from the system. The address of the task control block is an output of the ATTACH macro instruction.

[ ] indicates optional name.

DETACH -- Delete a Subtask (VS2)

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 operand. 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 operand 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:

[symbol]	DETACH	tcb location address [,STAE=$\left\{\begin{matrix}\text{YES}\\ \underline{\text{NO}}\end{matrix}\right\}$]

tcb location address
  is the virtual storage address of a fullword on a fullword boundary. The fullword contains the address of the task control block for the subtask to be removed from the system.

STAE=
  YES
    indicates that the exit routine specified in a STAE macro instruction issued by the subtask is to be given control if the subtask is scheduled for abnormal termination while it is being detached. If a retry routine is specified by the STAE exit routine, it is not given control.

  NO
    indicates that the exit routine specified in the STAE macro instruction will not be given control if the subtask is scheduled for abnormal termination (ABEND) while it is being detached. If neither YES nor NO is specified, NO is assumed.

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

Hexadecimal Code	Explanation
00	Successful completion.
04	Incomplete subtask detached; STAE exit taken.

DOM -- Delete Operator Message (MCS with DIDOCS only)

The DOM macro instruction deletes a message or group of messages from display on the display operator console. When a program no longer requires that a message be displayed, issue a DOM macro instruction to delete the message.

Depending on the timing of the DOM relative to the WTO(R), the message may or may not be displayed. If the message is being displayed, it is removed when space is required for other messages.

When a WTO or WTOR macro instruction is 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 should be coded using the identification number that was returned in general register 1.

The DOM macro instruction is written as follows:

```
┌───────────┬──────┬────────────────────┐
│ [symbol]  │ DOM  │ {MSG=register    } │
│           │      │ {MSGLIST=address } │
└───────────┴──────┴────────────────────┘
```

MSG=
: specifies a general register from 1 through 12 that contains the 24-bit, right-justified identification number of the message to be deleted. Use this operand to delete a single message. If you use register 1, the macro expansion is shortened by two bytes.

MSGLIST=
: specifies the address 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 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.

[ ] indicates optional name; select one from the vertical stack within { }.

DXR -- Divide Extended Register

Use the DXR macro instruction 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:

[symbol]	DXR	reg1,reg2

reg1
    is the register that contains the dividend. The quotient is placed in this register; the remainder is discarded.

reg2
    is the register that contains the divisor.

Notes: Following is a list of limitations that apply to both the reg1 and the reg2 operand:

- Registers 0 and 4 are the only registers that can be specified. However, you can specify them in either order.
- Specify registers as decimal digits 0 or 4 or as symbols that have been equated to these decimal digits.
- Never code these registers within parentheses.

[ ] indicates optional name.

ENQ -- Request Control of a Serially Reusable Resource

ENQ requests the control program to assign control of one or more (up to 255) serially reusable resources to the active task. If any of the resources are not available, the active task is 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 ENQ by a pair of names, the <u>qname</u> and the <u>rname</u>. The control program does not correlate the names with the actual resource. The relation of the name to the actual resource is your responsibility; 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, or if resource input addresses are incorrect, the task is abnormally terminated.

The standard form of the ENQ macro is written as follows. Information about the list and execute forms follows this description.

```
┌───────────┬──────┬──────────────────────────────────────────────────┐
│ [symbol]  │ ENQ  │(qname address,rname address,┌E┐,[rname length], │
│           │      │                             └S┘                 │
│           │      │┌SYSTEM┐,...)  ┌,RET=TEST┐                        │
│           │      │└STEP  ┘       │,RET=USE │                        │
│           │      │               │,RET=HAVE│                        │
│           │      │               └,RET=CHNG┘                        │
└───────────┴──────┴──────────────────────────────────────────────────┘
```

qname address
: is the address in virtual storage of an eight-character name. Every program issuing a request for a serially reusable resource must use the same qname and rname to represent the resource. The name should not start with SYS, so that it will not conflict with system names.

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

[ ] indicates optional name or operand; select none or one from vertical stack within [ ]; ,... indicates that more than one resource may be specified in the ENQ macro instruction, to a maximum of 255; ____ indicates an assumed value.

E
: is written as shown. It specifies that the request is for exclusive control of the resource. If you omit the operand, a request for exclusive control is assumed. If the resource is modified while under control of the task, the request must be for exclusive control.

S
: is written as shown. It specifies that the request is for shared control of the resource. If the resource is not modified while under control of the task, the request should be for shared control.

rname length
: is the length of the rname described above. If you omit the operand, 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 address designated above.

STEP
: is written as shown. It specifies that the resource is used only within the job step of the issuing program, and that a request for the same qname and rname from a program in another job step denotes a different resource. This option is assumed if the operand is omitted.

SYSTEM
: is written as shown. It specifies that the resource may be used by programs of more than one job step, and that requests for the same qname and rname from programs of other job steps in the system denote the same resource.

  Because SYSTEM and STEP are opposite in meaning, both cannot refer to the same resource. If two macro instructions specify the same qname and rname, but one specifies SYSTEM and the other specifies STEP, they are treated as requests for different resources. Conversely, when one resource is used by a single job step and another is used by several job steps, the same qname and rname can be used for both.

RET=
: specifies a conditional request for all of the resources named in the ENQ macro instruction. If the operand is omitted, the request is unconditional. The results of a conditional request are indicated by the return codes described in Figure 54; the types of conditional requests follow.

  TEST - tests the availability of the resources but does not request control of the resources.

  USE - specifies that control of the resources 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.

  HAVE - specifies that control of the resources is requested only if a request has not been made previously for the same task.

  CHNG - specifies that the status of the resource specified is to be changed from shared to exclusive control.

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 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 55. 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 54.

Code	Meaning			
	RET=TEST	RET=USE	RET=HAVE	RET=CHNG
0	The resource is immediately available.	Control of the resource has been assigned to the active task.		The status of the resource has been changed to exclusive.
4	The resource is not immediately available.		---	The status cannot be changed to shared.
8	A previous request for control of the same resource has been made for the same task. Task has control of resource.			The resource has not been queued.
12	Resource is permanently unavailable.			
20	A previous request for control of the same resource has been made for the same task. Task does not have control of resource.			Not used.

Figure 54. ENQ return codes

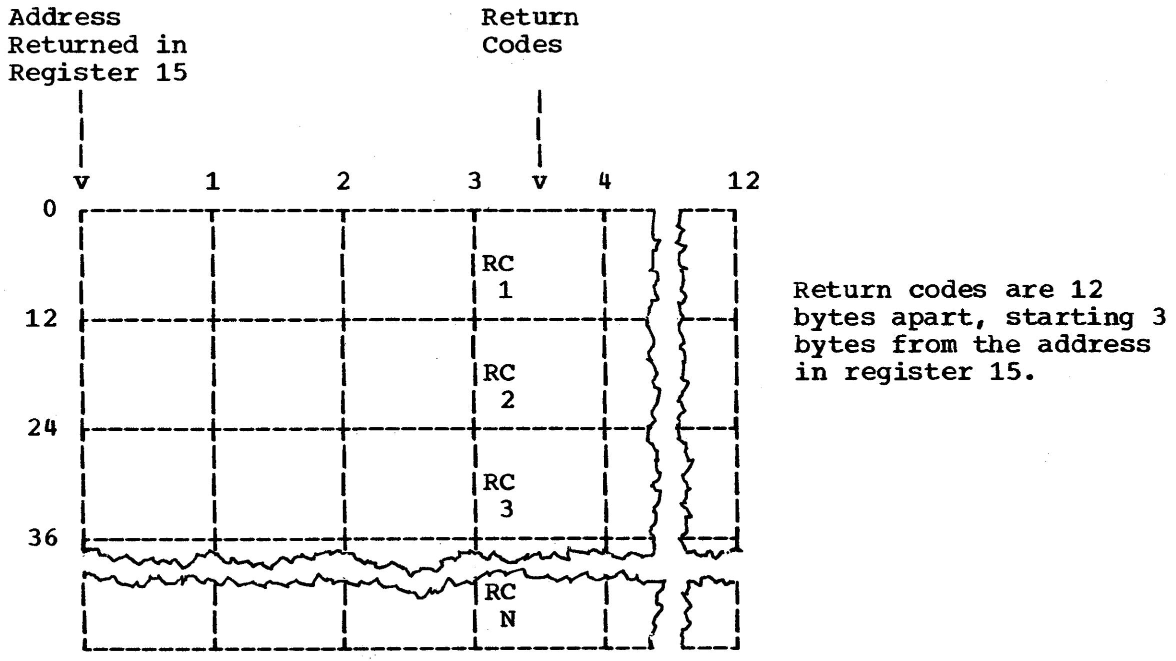

Address Returned in Register 15
Return Codes
v
1
2
3
v
4
12
0
12
24
36
RC 1
RC 2
RC 3
RC N
Return codes are 12 bytes apart, starting 3 bytes from the address in register 15.

Figure 55. Return code area used by ENQ

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 and rname combinations in the list form of the ENQ macro instruction must be equal to the maximum number of qname and rname combinations in any execute form of the macro instruction that refers to that list form.

The description of the standard forπ of ENQ provides the explanation the function of each operand. The description of the standard form also indicates which operands are always optional and which are required in at least one of the pair of list and execute forms. The format description below indicates the optional and required operands in the list form only.

```
┌──────────┬─────┬──────────────────────────────────────────────────────┐
│ [symbol] │ ENQ │ ([qname address],[rname address],┌E┐,[rname length],│
│          │     │                                  └S┘                │
│          │     │ ┌SYSTEM┐,...) ┌,RET=HAVE┐,MF=L                       │
│          │     │ │STEP  │      │,RET=TEST│                            │
│          │     │ └      ┘      │,RET=USE │                            │
│          │     │               └,RET=CHNG┘                            │
└──────────┴─────┴──────────────────────────────────────────────────────┘
```

symbol
    is any symbol valid in the assembler language.

address
    is any address that may be written in an A-type address constant.

length
    is any absolute expression valid in the assembler language.

MF=L
    indicates the list form of the ENQ macro.

[ ] indicates optional name or operand; ,... indicates that more than one qname and rname combination with associated options can be coded; select one or none from vertical stack within [ ]; ____ indicates an assumed value.

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 description of the standard form of ENQ provides the explanation the function of each operand. The description of the standard form also indicates which operands are always optional and which operands are required in at least one of the pair of list and execute forms. The format description below indicates the optional and required operands in the execute form only.

```
┌───────────┬──────┬──────────────────────────────────────────────────────┐
│ [symbol]  │ ENQ  │[([qname address],[rname address],┌E┐,[rname length],│
│           │      │                                   └S┘                │
│           │      │┌SYSTEM┐,...)] ┌,RET=HAVE┐                            │
│           │      ││STEP  │       │,RET=TEST│                            │
│           │      │└      ┘       │,RET=USE │                            │
│           │      │               │,RET=CHNG│                            │
│           │      │               └,RET=NONE┘                            │
│           │      │,MF=(E,{control program list address})                │
│           │      │       {            (1)             }                 │
└───────────┴──────┴──────────────────────────────────────────────────────┘
```

symbol
    is any symbol valid in the assembler language.

address
    is any address that is valid in an RX-type instruction, or one of general registers 2 through 12, previously loaded with the indicated address. The register may be designated symbolically or with an absolute expression; always code it within parentheses.

length
    is any absolute expression that is valid in the assembler language, or one of general registers 2 through 12, previously loaded with the indicated value. The register may be designated symbolically or with an absolute expression, and is always coded within parentheses.

RET=NONE
    specifies an unconditional request for control of all of the resources. The request is processed as though no RET operand had been coded.

MF=(E, {control program list address})
       {   (1)   }
    indicates the execute form of the macro instruction using a remote control program parameter list. The address of the control program parameter list can be coded as described under "address," or can be loaded into register 1, in which case code MF=(E,(1)).

[ ] indicates optional name or operand; select one from vertical stack within { }; ,... indicates that more than one qname and and rname combination with associated options can be coded; select one or none from vertical stack within [ ]; ____ indicates an assumed value.

FREEMAIN -- Release Allocated 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 if the specified area or subpool is not currently allocated to the active task. Register 15 is set to 0 to indicate successful completion.

The standard form of the FREEMAIN macro instruction is written as shown in the format description below. The operand combinations in the shaded area of the format description below must not be used in a VS1 system; the job step would be abnormally terminated.

```
[symbol]   FREEMAIN   / E,LV=number,A=address[,SP=number]  \
                      | L,LA=address,A=address[,SP=number] |
                      | R,SP=number                        |
                      | R,SP=(0)                           |
                      < R,LV=(0),A=address                 >
                      | R,LV=(0),A=(1)                     |
                      | R,LV=number,A=address[,SP=number]  |
                      | R,LV=number,A=(1)[,SP=number]      |
                      \ V,A=address[,SP=number]            /

Note: Only those operand combinations indicated above are valid.
```

E

(element) written as shown; specifies release of a single area of virtual storage allocated from the subpool indicated by the SP operand. The length of the virtual storage area is indicated by the LV operand; the address of the virtual storage area is provided at the address indicated in the A operand.

L

(list) written as shown; specifies release of one or more areas of virtual storage from the subpool indicated by the SP operand. The length of each virtual storage area is indicated by the values in a list beginning at the address specified in the LA operand. The address of each of the virtual storage areas must be provided in a corresponding list whose address is specified in the A operand. All virtual storage areas must start on a double word boundary.

R

(register) written as shown; specifies release of one area of virtual storage from the subpool indicated by the SP operand, or specifies release of the entire subpool indicated by the SP operand. If the release is not for the entire subpool, the address of the virtual storage area is indicated by the A operand. The length of the area is indicated by the LV operand. The virtual storage area must start on a doubleword boundary.

V

(variable) written as shown; specifies release of one area of virtual storage from the subpool indicated by the SP operand. The address and length of the virtual storage area are provided at the address specified in the A operand.

[ ] indicates optional name or operand; select one from vertical stack within { }.

LV=
is 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 higher 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 operand is invalied).

A=
is 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 completely within the area and must not begin in the first two words of the first area. If E or R is designated, one word, which contains the address of the virtual storage area to be released, is required. If V 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 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 is coded, any of the registers 1 through 12 can be designated, in which case the address of the virtual storage area, <u>not</u> 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.

LA=
is 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 operand. 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 operand.

SP=
if the SP operand is optional (shown within brackets), it specifies the subpool number of the virtual storage area to be released. The subpool number can be between 0 and 127. If the SP operand is optional and is omitted, subpool 0 is assumed. If the SP operand 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. 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.

FREEMAIN -- List Form

Use the list form of the FREEMAIN macro instruction to construct a nonexecutable control program parameter list. Do not use the list and execute forms of the FREEMAIN macro instruction with the R-type (register) of the macro instruction.

The description of the standard form of FREEMAIN explains the function of each operand. The description of the standard form also indicates which operands are always optional and which are required in at least one of the pair of list and execute forms. The operand combinations in the shaded area of the format description may only be used in a VS2 system. The format description below indicates the optional and required operands in the list form only.

```
┌───────────┬───────────┬──────────────────────────────────────────────┐
│ [symbol]  │ FREEMAIN  │ ⎧[E][,LV=number][,A=address][,SP=number]  ⎫ │
│           │           │ ⎨[L][,LA=address][,A=address][,SP=number] ⎬ │
│           │           │ ⎩[V][,A=address][,SP=number]              ⎭ │
│           │           │ MF=L                                         │
├───────────┴───────────┴──────────────────────────────────────────────┤
│ Note: Only those operand combinations indicated above are valid.    │
└──────────────────────────────────────────────────────────────────────┘
```

symbol
  is any symbol valid in the assembler language.

address
  is any address that may be written in an A-type address constant.

number
  is any absolute expression valid in the assembler language.

MF=L
  indicates the list form of the FREEMAIN macro instruction.

[ ] indicates optional name or operand; select one from vertical stack within { }.

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 list and execute forms of FREEMAIN cannot be used with the register (R) type of the macro instruction.

The description of the standard form of FREEMAIN explains the function of each operand. The description of the standard form also indicates which operands are always optional and which are required in at least one of the pair of list and execute forms. The operand combinations in the shaded area of the format description may only be used in a VS2 system. The format description below indicates the optional and required operands in the execute form only.

```
┌──────────┬──────────┬─────────────────────────────────────────┐
│ [symbol] │ FREEMAIN │ ⎧[E][,LV=number][,A=address][,SP=number] ⎫│
│          │          │ ⎨[L][,LA=address][,A=address][,SP=number]⎬│
│          │          │ ⎩[V][,A=address][,SP=number]             ⎭│
│          │          │ ,MF=(E,⎧control program list address⎫)   │
│          │          │        ⎩            (1)             ⎭    │
├──────────┴──────────┴─────────────────────────────────────────┤
│ Note: Only those operand combinations indicated above are valid.│
└───────────────────────────────────────────────────────────────┘
```

symbol
    is any symbol valid in the assembler language.

address
    is any address that is valid in an RX-type instruction, or one of general registers 2 through 12, previously loaded with the indicated address. You may designate the register symbolically or with an absolute expression; always code it within parentheses.

number
    is any absolute expression that is valid in the assembler language, or one of general registers 2 through 12, previously loaded with the indicated value. You may designate the register symbolically or with an absolute expression; always code it within parentheses.

```
MF=(E,⎧control program list address⎫)
      ⎩            (1)             ⎭
```
    indicates the execute form of the macro instruction and specifies the address of a remote control program parameter list constructed by the list form of FREEMAIN. In the remote control program parameter list, the operands for the execute form of FREEMAIN overlay the operands specified by the list form.

[ ] indicates optional name or operand; select one from vertical stack within { }.

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 to the job step. The virtual storage areas are released when the task assigned ownership terminates, or through the use of the FREEMAIN macro instructions.

The control program does not use the virtual storage area of the address in register 13 as a save area when processing release requests, if R is coded.

The standard form of the GETMAIN macro instruction is written as shown in the format description below. The operand combinations in the shaded area of the format description below must not be used in a VS1 system; the job steps would be abnormally terminated.

```
┌─────────┬─────────┬──────────────────────────────────────────────────────────────────┐
│[symbol] │GETMAIN  │/EC,LV=number,A=address[,SP=number]   ┌                  ┐ \ │
│         │         ││EU,LV=number,A=address[,SP=number]   │,BNDRY=┌DBLWD┐   │ │ │
│         │         │\LC,LA=address,A=address[,SP=number]  │       └PAGE ┘   │ / │
│         │         │/LU,LA=address,A=address[,SP=number]  └                  ┘ \ │
│         │         │\R,LV=number,[,SP=number]                                  │ │
│         │         │/R,LV=(0)                                                  │ │
│         │         ││VC,LA=address,A=address[,SP=number]  [,BNDRY=┌DBLWD┐]    │ │
│         │         │\VU,LA=address,A=address[,SP=number]  └       └PAGE ┘┘    / │
├─────────┴─────────┴──────────────────────────────────────────────────────────────────┤
│Note: Only those operand combinations indicated above are valid.                      │
└──────────────────────────────────────────────────────────────────────────────────────┘
```

E
: (element) written as shown; specifies a request for a single area of virtual storage from the subpool indicated by the SP number, having a length indicated by the LV operand. The address of the allocated virtual storage area is returned at the address indicated in the A operand.

L
: (list) written as shown; specifies a request for one or more areas of virtual storage from the subpool indicated by the SP number. The length of each virtual storage area is indicated by the values in a list beginning at the address specified in the LA operand. The address of each of the virtual storage areas is returned in a list beginning at the address specified in the A operand. No virtual storage is allocated unless all of the requests in the list can be satisfied.

R
: (register) written as shown; specifies a request for a single area of virtual storage to be allocated from the indicated subpool, and to have a length indicated by the LV operand. The address of the allocated virtual storage area is returned in register 1. If R is designated, the requests are unconditional; a request for more virtual storage than is available results in abnormal termination of the task.

[ ] indicates optional name or operand; select one from vertical stack within { }; ____ indicates an assumed value.

V

(variable) written as shown; specifies a request for a single area of virtual storage to be allocated from the subpool indicated by the SP number, and to have a length to be between two values at the address specified in the LA operand. The address and actual length of the allocated virtual storage area are returned by the control program at the address indicated in the A operand.

C

(conditional) written as shown; specifies that the request is conditional and that the task is not to be abnormally terminated if more virtual storage is requested than is available. If the request is staisfied, register 15 contains a return code of 0; if not satisified, the return code is 4.

U

(unconditional) written as shown; specifies that the request is unconditional. An unconditional request for more virtual storage than is available will result in abnormal termination of the requesting task.

LV=

is 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.

LA=

is the virtual storage address of consecutive fullwords starting on a fullword boundary. Each fullword must contain the required length in the low-order three bytes, with the high-order byte set to 0. The lengths should be multiples of 8; if they are not, the control program uses the next higher multiple of 8. If V was coded, two words are required. The first word contains the minimum length required, the second word contains the maximum length. If L was coded, one word is required for each virtual storage area requested; the high-order bit is 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 operand.

A=

is 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 the low-order three bytes. 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 operand.

SP=

is the number of the subpool from which the virtual storage area is to be allocated. The number must be between 0 and 127. If the operand is omitted, subpool 0 is specified.

BNDRY=

is the type of alignment required for the start of the requested area. DBLWD indicates a doubleword boundary; PAGE indicates alignment with the start of a virtual page (2K boundary). If BNDRY= is omitted, DBLWD is assumed. BNDRY= is not valid for R-type (register) GETMAINs.

After execution of the GETMAIN requests, the return code in register 15 is as follows:

Hexadecimal Code	Meaning
00	The virtual storage requested was allocated.
04	No vritual storage was allocated (conditional form only).

Note: A request for zero bytes or an unconditional request for more virtual storage than is available results in abnormal termination of the job step.

GETMAIN -- List Form

Use the list form of the GETMAIN macro instruction to construct a control program parameter list. The list and execute forms of GETMAIN cannot be used with the R-type (register) of the macro instruction.

The description of the standard form of GETMAIN explains the function of each operand. The description of the standard form also indicates which operands are always optional and which are required in at least one of the pair of list and execute forms. The operand combinations in the shaded area of the format description may only be used in a VS2 system. The format description below indicates the optional and required operands in the list form only.

```
┌───────────┬───────────┬──────────────────────────────────────────────────┐
│           │           │ ⎛[EC][,LV=number]⎞                               │
│ [symbol]  │ GETMAIN   │ ⎜[EU][,LV=number]⎟[,A=address][,SP=number]       │
│           │           │ ⎜[LC][,LA=address]⎟                              │
│           │           │ ⎨[LU][,LA=address]⎬                              │
│           │           │ ⎜[VC][,LA=address]⎟                              │
│           │           │ ⎝[VU][,LA=address]⎠                              │
│           │           │                                                  │
│           │           │ [,BNDRY={DBLWD}],MF=L                            │
│           │           │         {PAGE }                                  │
├───────────┴───────────┴──────────────────────────────────────────────────┤
│ Note: Only those operand combinations indicated above are valid.         │
└──────────────────────────────────────────────────────────────────────────┘
```

symbol
    is any symbol valid in the assembler language.

address
    is any address that may be written in an A-type address constant.

number
    is any absolute expression valid in the assembler language.

MF=L
    indicates the list form of the GETMAIN macro instruction.

[ ] indicates optional name or operand; select one from vertical stack within { }; ____ indicates an assumed value.

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 list and execute forms of GETMAIN cannot be used with the R-type (register) of the macro instruction.

The description of the standard form of GETMAIN explains the function of each operand. The description of the standard form also indicates which operands are always optional and which are required in at least one of the pair of list and execute forms. The operand combinations in the shaded area of the format description may only be used in a VS2 system. The format description below indicates the optional and required operands in the execute form only.

```
┌──────────┬──────────┬──────────────────────────────────────────────────────┐
│ [symbol] │ GETMAIN  │ ⎧[EC][,LV=number] ⎫                                  │
│          │          │ ⎪[EU][,LV=number] ⎪[,A=address][,SP=number]          │
│          │          │ ⎨[LC][,LA=address]⎬                                  │
│          │          │ ⎪[LU][,LA=address]⎪                                  │
│          │          │ ⎪[VC][,LA=address]⎪                                  │
│          │          │ ⎩[VU][,LA=address]⎭                                  │
│          │          │                                                      │
│          │          │ [,BNDRY={DBLWD}]                                     │
│          │          │         {PAGE }                                      │
│          │          │                                                      │
│          │          │ ,MF=(E,{control program list address})               │
│          │          │        {            (1)              }               │
├──────────┴──────────┴──────────────────────────────────────────────────────┤
│ Note: Only those operand combinations indicated above are valid.           │
└────────────────────────────────────────────────────────────────────────────┘
```

symbol
    is any symbol valid in the assembler language.

address
    is any address that is valid in an RX-type instruction, or one of general registers 2 through 12, previously loaded with the indicated address. You may designate the register symbolically or with an absolute expression; always code it within parentheses.

number
    is any absolute expression that is valid in the assembler language, or one of general registers 2 through 12, previously loaded with the indicated value. The register may be designated symbolically or with an absolute expression, and is always coded within parentheses.

```
MF=(E,{control program list address})
      {            (1)              }
```
    indicates the execute form of the macro instruction using a remote control program parameter list. The address of the control program parameter list can be coded as described under "address," or can be loaded into register 1, in which case MF=(E,(1)) should be coded.

[ ] indicates optional name or operand; select one from vertical stack within { }; ____ indicates an assumed value.

GTRACE -- Record Trace Data

Use the GTRACE macro instruction to record data in a trace data set, using GTF (generalized trace facility). This data set may later become input to an editing function provided by HMDPRDMP service aid. An optional parameter allows you to specify a specific format routine (user written if desired) to process the record after the trace output is edited.

To use GTRACE, GTF must be active and ready to accept data from the problem program. Also, the trace data set must be identified in the GTF job control statements.

The standard form of the GTRACE macro instruction is written as shown below. Information about the list and execute forms follows this description.

[symbol]	GTRACE	DATA=address,LNG=number,ID=number [,FID=number]

DATA=
is the virtual storage address of the data to be recorded.

LNG=
is the number of bytes of data to be recorded. Specify any number from 1 to 256.

ID=
is the identifier to be associated with the record. ID values are assigned as follows:

0-1023	user events
1024-4095	reserved

FID=
is the format identifier indicating the format routine used to process the record when the trace output is edited by HMDPRDMP. You may provide your own routine to handle this data. Format identifier values are:

0	hexadecimal dump of entry
1-80	user format identifiers
81-255	reserved

If the FID parameter is omitted, 0 is assumed.

The format identifier must be converted to hexadecimal. If it is not 0, it is appended to the name HMDUSR to form the name of the format routine used by HMDPRDMP to process the record. For example, FID=50 converts to X'32'. Module HMDUSR32 is used by HMDPRDMP to process the trace record.

Formatting routines must be in SYS1.LINKLIB or in a private library defined in a JOBLIB or STEPLIB DD statement for HMDPRDMP.

A return code is placed in register 15 when control is returned to the problem program.

[ ] indicates optional name or operand.

Hexadecimal Code	Explanation
00	Successful completion.
04	GTF not active or not accepting problem program entries.
08	Length specified in LNG parameter is greater than 256.
0C	Invalid data address.
10	FID value is greater than 255.
14	Value of ID parameter is greater than 1023.
18	Buffers are full; record was not placed in buffer.
1C	Invalid address of parameter list.
20	Data paged-out, cannot be gathered.

GTRACE -- List Form

The list form of the GTRACE macro instruction constructs a control program parameter list. Use the list form of GTRACE to pass address parameters in a parameter list to the control program. This parameter list can then be referred to by issuing the execute form of GTRACE.

The description of the standard form of GTRACE explains the function of each operand. The format description below indicates the optional and required operands for the list form only.

[symbol]	GTRACE	[DATA=address][,LNG=length] [,FID=number],MF=L

address
    is any address value that can be expressed as an A-type address constant.

length
    is any number from 1 to 256.

number
    is any number from 0 to 80.

MF=L
    indicates the list form of the macro instruction.

Note: The ID parameter is not valid in the list form of GTRACE.

[ ] indicates optional name or operand.

GTRACE -- Execute Form

The execute form of the GTRACE macro instruction uses the remote control program parameter list created by the list form of GTRACE. The description of the standard form explains the function of each operand. The format description below indicates the operands for the execute form only.

```
┌──────────┬─────────┬────────────────────────────────────────────┐
│ [symbol] │ GTRACE  │ ID=value,MF=(E,{parameter list address})   │
│          │         │                {          (1-12)       }   │
│          │         │ [,DATA=address][,LNG=length]               │
│          │         │ [,FID=number]                              │
└──────────┴─────────┴────────────────────────────────────────────┘
```

value
    is any number from 0 to 1023.

```
MF=(E,{parameter list address})
      {        (1-12)         }
```
    indicates the execute form of the macro instruction using a remote control program parameter list. If you load the address of the list into register 1, code MF=(E,(1)). If the address is not loaded into register 1, code it as any address that is valid in an RX-type instruction, or as a register 2-12, previously loaded with the address. You may designate the register symbolically or with an absolute expression. Always code the register value in parentheses.

address
    is any address that can be expressed as an A-type address constant.

length
    is any number from 1 to 256.

number
    is any number from 0 to 80.

[ ] indicates optional name or operand; select one from vertical stack within { }.

IDENTIFY -- Add an Entry Point

Use the IDENTIFY macro instruction to add an entry point 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 any task within the partition/region.
- The last load module given control, if control was passed to the load module using a LINK, ATTACH, or XCTL macro instruction.
- The first load module of any task, if it is still in control.

The IDENTIFY macro instruction may not be issued by an asynchronous exit routine. The routine associated with the entry point must be reenterable.

In VS1, IDENTIFY may not be issued by a routine entered at an added entry point. The added entry point can be used only in an ATTACH macro instruction.

The IDENTIFY macro instruction is written as follows:

```
┌────────┬────────┬───────────────────────────────────────────────────┐
│[symbol]│IDENTIFY│{EP=symbol            },ENTRY=entry point address   │
│        │        │{EPLOC=address of name}                             │
└────────┴────────┴───────────────────────────────────────────────────┘
```

EP=
: is the name of the entry point. The name does not have to correspond to any name or symbol in the load module, and must not correspond to any name, alias, or added entry point for a load module in the resident reenterable module area, LPAQ area, or the job pack area of the job step.

EPLOC=
: is the address of the entry point name described under EP. Pad the name with blanks to eight bytes, if necessary.

ENTRY=
: is the virtual storage address of the entry point being added.

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

Hexadecimal Code	Meaning
00	Successful completion.
04	Entry point name and address already exist.
08	Entry point name duplicates the name of a load module currently in virtual storage; entry point was not added.
0C	Entry point address is not within an eligible load module; entry point was not added.
10	Issued by an asynchronous exit routine; the entry point was not added.
14	An IDENTIFY macro instruction was previously issued using the same entry point name but a different address; this request was ignored.

[ ] indicates optional name; select one from vertical stack within { }.

LINK -- Pass Control to a Program in Another Load Module

Use the LINK macro instruction to pass control to a specified entry point in another load module; the entry point 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. (Refer to the Services section of this publication, for a discussion of the use of an existing copy of the load module.)

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 shown below. Information about the list and execute forms follows this description.

```
┌──────────┬──────┬──────────────────────────────────────────────┐
│ [symbol] │ LINK │ ⎧EP=symbol           ⎫    [,DCB=dcb address] │
│          │      │ ⎨EPLOC=address of name⎬                      │
│          │      │ ⎩DE=address of list entry⎭                   │
│          │      │  [,PARAM=(addresses)][,VL=1][,ID=number]     │
└──────────┴──────┴──────────────────────────────────────────────┘
```

EP=
    is the entry point name in the program to be given control.

EPLOC=
    is the address of the entry point name described above. Pad the name with blanks to eight bytes, if necessary.

DE=
    is the address of the name field of a 58-byte (60 bytes in VS2) list entry for the entry point name. The list entry is constructed using the BLDL macro instruction using a length specification of 58 or 60 bytes. The DCB operand must indicate the same data control block used in BLDL.

DCB=
    is the address of the data control block for the partitioned data set containing the entry point name described above.

    If the DCB operand 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 operand 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.

[ ] indicates optional name or operand; select one from vertical stack within { }.

PARAM=

is one or more address parameters, separated by commas, 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 operand is omitted, register 1 is not altered.)

VL=1

is written as shown. It can be designated only if PARAM is designated, and should be used 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=

specifies a decimal integer with a maximum value of $2^{16}-1$. The last fullword of the macro expansion is a NOP instruction containing the ID value in bytes 3 and 4. This operand is useful for debugging purposes only.

LINK -- List Form

Two parameter lists are used in a LINK 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 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 description of the standard form of LINK explains the function of each operand. The description of the standard form also indicates which operands are always optional and which are required in at least one of the pair of list and execute forms. The format description below indicates the operands in the list form only.

```
┌───────────┬───────┬───────────────────────────────────────────────┐
│ [symbol]  │ LINK  │ ⎧EP=symbol                  ⎫[,DCB=dcb address] │
│           │       │ ⎨EPLOC=address of name      ⎬                   │
│           │       │ ⎩DE=address of list entry   ⎭                   │
│           │       │ ,SF=L                                           │
└───────────┴───────┴───────────────────────────────────────────────┘
```

symbol
: is any symbol valid in the assembler language.

address
: is any address that may be written in an A-type address constant.

SF=L
: indicates the list form of the LINK macro instruction.

[ ] indicates optional name or operands; select one or none from vertical stack within [ ].

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, operands that require use of the other parameter list cause that list to be constructed inline as part of the macro expansion.

The description of the standard form of LINK explains the function of each operand. The description of the standard form also indicates which operands are always optional and which are required in at least one of a pair of list and execute forms. The format description below indicates the operands in the execute form only.

```
┌──────────┬──────┬──────────────────────────────────────────────────┐
│          │      │ ⎧EP=symbol                 ⎫[,DCB=dcb address]  │
│ [symbol] │ LINK │ ⎨EPLOC=address of name     ⎬                    │
│          │      │ ⎩DE=address of list entry  ⎭                    │
│          │      │ [,PARAM=(addresses)][,VL=1][,ID=number]          │
│          │      │ ⎧,MF=(E,⎧problem program list address⎫)⎫         │
│          │      │ ⎪       ⎩           (1)              ⎭ ⎪         │
│          │      │ ⎨,SF=(E,⎧control program list address⎫)⎬         │
│          │      │ ⎪       ⎩           (15)             ⎭ ⎪         │
│          │      │ ⎪,MF=(E,⎧address⎫),SF=(E,⎧address⎫)    ⎪         │
│          │      │ ⎩       ⎩  (1)  ⎭        ⎩ (15)  ⎭     ⎭         │
└──────────┴──────┴──────────────────────────────────────────────────┘
```

symbol
    is any symbol valid in the assembler language.

address
    is any address that is valid in an RX-type instruction, or one of general registers 2 through 12, previously loaded with the indicated address. The register may be designated symbolically or with an absolute expression; always code it within parentheses.

number
    is any absolute expression that is valid in the assembler language.

```
MF=(E,⎧problem program list address⎫)
      ⎩           (1)              ⎭
```

indicates the execute form of the macro instruction using a remote problem program parameter list. Any control program parameters specified are provided in a control program parameter list expanded inline. The address of the problem program parameter list can be coded as described under "address," or can be loaded into register 1, in which case code MF=(E,(1)). If you code the PARAM operand, the addresses are placed in contiguous fullwords at the user problem program list address.

```
SF=(E,⎧control program list address⎫)
      ⎩           (15)             ⎭
```

indicates the execute form of the macro instruction using a remote control program parameter list. Any problem program parameters specified are provided in a problem program parameter list expanded inline. The address of the control program parameter list can be coded as described under "address," or can be loaded into register 15, in which case code SF=(E,(15)).

[ ] indicates optional name or operand; select one from vertical stack within { }; select one or none from vertical stack within [ ].

```
MF=(E,{address}),SF=(E,{address})
      {   (1)  }         {  (15) }
```

indicates the execute form of the macro instruction using <u>both</u> a remote problem program parameter list and a remote control program parameter list. The addresses of the parameter lists are coded or loaded into registers 1 and 15, as explained above.

Select one from vertical stack within { }.

<u>LOAD -- Bring a Load Module Into Virtual Storage</u>

Use the LOAD macro instruction to bring the load module containing the specified entry point into virtual storage, if a usable copy is not available in virtual storage. (Refer to "Using an Existing Copy" for a discussion of the use of an existing copy of the load module.) The responsibility count for the load module is increased by one. On output, the high-order byte of register 1 contains the authorization code of the loaded module and the low three bytes contain the module's length. Control is <u>not</u> passed to the load module; instead, the virtual storage address of the designated entry point is returned in register 0. The load module remains in virtual storage until the responsibility count is reduced to 0 through task terminations or until the effects of all outstanding LOAD requests for the module have been canceled (using the DELETE macro instruction), <u>and</u> there is no other requirement for the module.

The entry point 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 point cannot be located, the task is abnormally terminated.

```
┌──────────┬──────┬─────────────────────────────────────────────────┐
│ [symbol] │ LOAD │ ⎧EP=symbol                ⎫[,DCB=dcb address]   │
│          │      │ ⎨EPLOC=address of name    ⎬                     │
│          │      │ ⎩DE=address of list entry ⎭                     │
└──────────┴──────┴─────────────────────────────────────────────────┘
```

EP=
: is the entry point name in the load module to be brought into virtual storage. Pad the name with blanks to eight bytes, if necessary.

EPLOC=
: is the virtual storage address of the entry point name described above. Pad the name with blanks to eight bytes, if necessary.

DE=
: is the address of the name field of a 58-byte list entry for the entry point name instruction. The list entry is constructed by a BLDL macro, using a length specification of 58 bytes. The DCB operand must indicate the same data control block used in the BLDL macro.

DCB=
: is the address of the data control block for the partitioned data set containing the entry point name described above.

  If the DCB operand is omitted or if DCB=0 is specified when LOAD is issued by the <u>job step task</u>, 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= operand is omitted or if DCB=0 is specified when LOAD is issued by a <u>subtask</u>, the data sets associated with one or more data control blocks referred to by previous ATTACH macros 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 LOAD had been issued by the job step task.

[ ] indicates optional name operand; select one from vertical stack within { }.

PGRLSE -- Release Virtual Storage Contents

The PGRLSE macro instruction makes all complete pages of real and external page storage wholly associated with the area of virtual address space specified available. The address space remains intact but its contents are forfeited. Use PGRLSE when a large area (one or more complete pages) of virtual storage within your program no longer has meaningful or significant contents.

When you issue a PGRLSE macro instruction, all complete pages of virtual storage between the low and high addresses specified are released. You can help reduce system overhead by releasing virtual storage when you no longer need it.

The standard form of the PGRLSE macro instruction is written as follows. Information about the list and execute forms follows this description.

```
┌──────────┬─────────┬───────────────────────────┐
│ [symbol] │ PGRLSE  │ LA={addr1 },HA={addr2 }   │
│          │         │    {(reg1)}    {(reg2)}   │
└──────────┴─────────┴───────────────────────────┘
```

LA=
: is the low address of the area to be released. Addr1 specifies the low address; reg1 indicates a general register containing the address. LA=(1) may not be specified.

HA=
: is the high address + 1 of the area to be released (low address + length of area). Addr2 specifies the high address + 1; reg2 indicates a general register containing the address. HA=(0) may not be specified.

Upon completion of PGRLSE, register 15 is set as follows:

Hexadecimal Code	Meaning
00	Successful completion.
04	Execution failed. The area specified, or a portion of it, is protected from the requesting program. Any valid portion of the area preceding the protected area is released.

[ ] indicates optional name; select one from vertical stack within { }.

PGRLSE -- List Form

Use the list form of the PGRLSE macro instruction to construct a control program parameter list.

The description of the standard form of PGRLSE explains the function of each operand. The description of the standard form also indicates which operands are always optional and which are required in at least one of the pair of list and execute forms. The following format description indicates the operands in the list form only. If LA or HA is coded, addresses must be specified; register notation cannot be used.

[symbol]	PGRLSE	MF=L[,LA=addr1][,HA=addr2]

symbol
  is any symbol valid in the assembler language.

MF=L
  indicates the list form of the PGRLSE macro instruction.

addr
  is any address that may be written in an A-type address constant.

[ ] indicates optional name or operands.

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 description of the standard form of PGRLSE explains the function of each operand. The description of the standard form also indicates which operands are always optional and which are required in at least one of the pair of list and execute forms. The format description below indicates the operands for the execute form only.

An execution error is indicated if the list address is outside of your partition, or if it is of a different storage protection key.

```
┌───────────┬──────────┬────────────────────────────────────────────────────────────┐
│ [symbol]  │ PGRLSE   │ MF=(E,{listaddr}) [,LA={addr1 }] [,HA={addr2 }]              │
│           │          │       { (reg3) }       {(reg1)}       {(reg2)}               │
└───────────┴──────────┴────────────────────────────────────────────────────────────┘
```

symbol
    is any symbol valid in assembler language.

```
MF=(E,{listaddr})
      { (reg3) }
```
    indicates the execute form of the macro instruction using a remote control program parameter list. The address can be any address that is valid in an RX-type instruction. Reg3 indicates a register (2-12) containing the address. Optimum performance occurs if a register is specified.

addr
    is any address that may be written as an A-type address constant. Reg indicates one of the general registers previously loaded with the indicated address. The register may be designated symbolically or with an absolute expression, and is always coded within parentheses. Neither LA=(1) nor HA=(0) may be specified.

[ ] indicates optional name or operand; select one from vertical stack within { }.

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 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.

Figure 58 shows the format of the event control block and its associated completion codes.

The POST macro is written as follows:

[symbol]	POST	ecb address[,completion code]

ecb address
    is the address of an event control block representing the event.

completion code
    is a value between 0 and $2^{24}-1$. If the completion code is not designated, 0 is assumed.

[ ] indicates optional name or operand.

RETURN -- Return Control

The RETURN macro instruction restores 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:

```
┌───────────┬─────────┬───────────────────────────────────┐
│ [symbol]  │ RETURN  │ [(reg1[,reg2])][,T] ┌,RC=number┐  │
│           │         │                     └,RC=(15)  ┘  │
└───────────┴─────────┴───────────────────────────────────┘
```

reg1,reg2
: is the range of registers to be restored from the save area pointed to by the address in register 13. The registers should be designated to cause the loading of registers 14, 15, 0 through 12 when used in an LM instruction. If you do not specify reg2, only the register designated by the reg1 operand is loaded. If you omit the operand, the contents of the registers are not altered. Do not code reg1 or reg2 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. It designates that a called program has executed a return to its caller. Do not specify this operand when returning control from an exit routine.

RC=
: is the return code to be passed to the calling program. The return code should have a maximum value of 4095; it is placed right-adjusted in register 15 before return is made. If you code RC=(15), it indicates that the return code has been previously loaded into register 15; in this case the contents of register 15 are not altered or restored from the save area. (If you omit this operand the contents of register 15 are determined by the reg1,reg2 operands.)

[ ] indicates optional name or operand; select one or none from vertical stack within [ ].

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 is written as follows:

[symbol]	SAVE	(reg1[,reg2]),[T][,identifier name]

reg1,reg2
: is the range of registers to be stored in the save area at the address contained in register 13. The registers should be designated so they are stored in the order 14, 15, 0 through 12 when used directly in an STM instruction. Do not specify register 13. The registers are stored in words 4 through 18 of the save area. If only one register is designated, only that register is saved.

T
: specifies that registers 14 and 15 are to be stored in words 4 and 5, respectively, of the save area. 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. The T operand permits you to save two noncontiguous sets of registers.

identifier name
: is an identifier to be associated with the SAVE macro instruction. The name may be up to 70 characters and may be a complex name. If an asterisk is coded, the identifier is the symbol 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 operand 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.

[ ] indicates optional name or operand.

SEGWT -- Load Overlay Segment and Wait

The SEGWT macro instruction causes the control program to load the specified segment and any segments in its path that are not part of a path already in virtual storage. Control is not passed to the specified segment; control is not returned to the segment issuing the SEGWT macro instruction until the requested segment is loaded. Refer to the publication Linkage Editor and Loader, for details on overlay operations. The SEGWT macro instruction cannot be used in an asynchronous exit routine.

The SEGWT macro instruction is written as follows:

[symbol]	SEGWT	external segment name

external segment name
is the name of a control section or entry point in the required segment. An exclusive reference is not allowed. The name does not have to be identified by an EXTRN statement.

[ ] indicates optional name.

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. The format of the dump is similar to the abnormal termination dump shown in the VS1 and VS2 Debugging Guides.

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

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

BLKSIZE must be 882 in VS1, and either 882 or 1632 in VS2. 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 is written as shown below. Information about the list and execute forms follows this description.

```
┌─────────┬─────┬───────────────────────────────────────────────────────┐
│[symbol] │SNAP │DCB=dcb address[,TCB=address]                          │
│         │     │[,ID=number][,SDATA=(code for control program blocks)] │
│         │     │[,PDATA=(code for problem program areas)]              │
│         │     │⎡,STORAGE=(starting address, ending address,...)⎤      │
│         │     │⎣,LIST=address of list                          ⎦      │
└─────────┴─────┴───────────────────────────────────────────────────────┘
```

DCB=
: is the address of the data control block for the data set that is to contain the dump. This DCB must be open before SNAP is executed. If the DCB is omitted or specified in register format, the DCB address will default to 0.

TCB=
: 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 task control block.

[ ] indicates optional name or operand; select one or none from vertical stack within [ ]; ,... indicates that more than one pair of starting and ending addresses can be specified.

ID=
is a number between 1 and 127. The number is printed in the identification heading associated with the dump. If specified in register format, the ID will default to 0.

SDATA=
one to four of the following sets of characters, written in any order and separated by commas. The characters are used to request the associated control program information:

Code	Fields Dumped
ALL	All of the following fields.
NUC	All of the control program nucleus except the trace table.
TRT	Trace Table. Ignored if GTF is active and was started with the S parameter specified (formatting of the GTF trace buffers suppressed).
CB	Task control block (TCB), active request blocks (RBs), job pack area control queue (JPACQ) and control blocks.
Q	Ignored.

PDATA=
one to five of the following sets of characters, written in any order and separated by commas. Use these characters to request the following problem program information:

Code	Fields Dumped
ALL	All of the following fields.
PSW	Program status word when the SNAP macro instruction was issued.
REGS	Contents of the general registers when the SNAP macro instruction was issued.
SA or SAH	SA - provides linkage information and a back trace through save areas. SA is selected if ALL is coded. SAH - only linkage information.
JPA or LPA or ALLPA	JPA - all virtual storage assigned to the job step. LPA - contents of the resident reenterable load module area. ALLPA - contents of both pack areas. ALLPA is selected if ALL is coded.
SPLS	All virtual storage assigned to job step.

STORAGE=
is one or more pairs of starting and ending addresses; the areas between the starting and ending addresses are dumped one fullword at a time. If the starting and ending addresses are not fullword multiples, the addresses are rounded down (starting) and up (ending) to a fullword.

The area to be dumped must be in your program's partition/region. If the addresses are not within the partition/region, a condition code of 00 is returned but the storage area is not dumped.

LIST=
the address of a list of starting and ending addresses of areas to be dumped. The addresses in the list are treated in the same manner as the addresses described for the STORAGE operand. The list must begin on a fullword boundary; each address in the list occupies one fullword. The high-order byte of each word containing the starting address of an area to be dumped must contain zeros or that pair will be skipped. The high-order bit (bit 0) of the fullword containing the last ending address in 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.
08	SVCDUMP issued by non-key 0, space unavailable, or dump taken by task with a job step as a subtask.
0C	Data control block type was not correct (DSORG, RECFM, MACRF, BLKSIZE, or LRECL field).

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 operand. 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 description of the standard form of the SNAP macro provides the explanation of the function of each operand. The description of the standard form also indicates which operands are always optional and which are required in at least one of the pair of list and execute forms. The format description below indicates the optional and required operands in the list form only.

```
┌─────────┬─────┬──────────────────────────────────────────────────────────┐
│[symbol] │SNAP │[DCB=address][,ID=number][,SDATA=(code)][,PDATA=(code)]   │
│         │     │┌,STORAGE=(address,address,...)┐,MF=L                    │
│         │     │└,LIST=address                 ┘                         │
└─────────┴─────┴──────────────────────────────────────────────────────────┘
```

symbol
    is any symbol valid in the assembler language.

address
    is any address that may be written in an A-type address constant. If the address is omitted, the default value is 0.

code
    is written as indicated in the description of the standard form of the macro instruction.

number
    is any absolute expression valid in the assembler language.

MF=L
    indicates the list form of the SNAP macro instruction.

[ ] indicates optional name or operand; select one or none from vertical stack within [ ]; ,... indicates that more than one pair of starting and ending addresses can be specified.

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 operands in the execute form of the macro instruction, the bit settings in the parameter list corresponding to the SDATA, PDATA, LIST, and STORAGE operands are not changed. However, if you code one or more of the SDATA, PDATA, LIST operands, the bit settings from the previous request are reset to zero, and only the areas requested in the current macro instruction are dumped.

The description of the standard form of SNAP explains the function of each operand. The description of the standard form also indicates which operands are always optional and which are required in at least one of the pair of list and execute forms. The format description below indicates the optional and required operands in the execute form only.

```
┌───────────┬───────┬──────────────────────────────────────────────────┐
│ [symbol]  │ SNAP  │ [DCB=address][,TCB=┌'S'     ┐][,ID=number]       │
│           │       │                    └address ┘                    │
│           │       │ [,PDATA=code][,SDATA=code]                       │
│           │       │ ┌,STORAGE=(address,address,...)┐                 │
│           │       │ └,LIST=address                 ┘                 │
│           │       │ ,MF=(E,┌control program list address┐)           │
│           │       │        └            (1)             ┘            │
└───────────┴───────┴──────────────────────────────────────────────────┘
```

symbol
  is any symbol valid in the assembler language.

address
  is any address that is valid in an RX-type instruction, or one of general registers 2 through 12, previously loaded with the indicated address. You may designate the register symbolically or with an absolute expression; always code it within parentheses. If the address is omitted, the default value is 0.

'S'
  is used to specify the task control block of the active task.

number
  is any absolute expression that is valid in the assembler language, or one of general registers 2 through 12, previously loaded with the indicated value. You may designate the register symbolically or with an absolute expression; always code it within parentheses.

code
  is written as indicated in the description of the standard format of the macro instruction.

MF=(E,{control program list address / (1)})
  indicates the execute form of the macro instruction using a remote control program parameter list. The address of the control program parameter list can be coded as described under "address," or can be loaded into register 1, in which case code MF=(E,(1)).

[ ] indicates optional name or operand; select one or none from vertical stack within [ ]; ,... indicates that more than one pair of starting and ending addresses can be specified. Select one from vertical stack within { }.

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, shown in Figure 56, contains the exit routine's address and a code indicating the interruption types specified in SPIE.

Any program issuing a SPIE macro instruction must restore the previous PICA before returning to the calling program. The previous PICA address is returned in register 1 after execution of SPIE; this address can be used to restore the PICA before returning control. If the SPIE macro instruction is the first issued in performance of the task, register 1 is set to 0 when control is returned. A SPIE macro instruction with no operands (a canceling SPIE) creates a null SPIE environment (program mask set to zero, no program interrupts intercepted, and no exit routine; however the PICA created by this SPIE is controlling the SPIE environment). To reestablish a previous SPIE, whether or not a canceling SPIE has been issued, issue the execute form of SPIE, specifying the address of the appropriate PICA. Issuing a canceling SPIE causes the address of the previous PICA to be returned.

In addition to the PICA, there is one PIE (program interrupt element) per task. The PIE is 32 bytes long, the first four bytes of which contain the address of the PICA. The format of the PIE is shown in Figure 54. The PIE is built when the first SPIE macro instruction is issued and remains in effect as long as the task is active.

The PICA address in the PIE is the address of the PICA 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 PSW 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 in Figure 57.

Bytes 1		3	2
0000	Program Mask	Exit Routine Address	Interruption Mask

Figure 56. Program interruption control area (PICA)

```
   0            1         2              3
   ┌────────────┬────────────────────────────────────┐
   │ Reserved   │           PICA Address             │
  4├────────────┴──────────┬─────────────────────────┤
   │     Old Program       │ (Interruption Codes)    │
   │     Status Word       └─────────────────────────┤
   │     (Resulting from the Interruption)           │
 12├─────────────────────────────────────────────────┤
   │                 Register 14                     │
 16├─────────────────────────────────────────────────┤
   │                 Register 15                     │
 20├─────────────────────────────────────────────────┤
   │                 Register 0                      │
 24├─────────────────────────────────────────────────┤
   │                 Register 1                      │
 28├─────────────────────────────────────────────────┤
   │                 Register 2                      │
 32└─────────────────────────────────────────────────┘
```

Figure 57. Program interruption element (PIE)

The standard form of the SPIE macro instruction is written as shown below. Information about the list and execute forms follows this description.

```
┌──────────┬──────┬───────────────────────────────────────────────┐
│ [symbol] │ SPIE │ [interruption exit address,(interruptions)]   │
└──────────┴──────┴───────────────────────────────────────────────┘
```

interruption exit address
: is the address of the exit routine to be given control when a program interruption of the type specified in the interruptions operand occurs.

interruptions
: is one or more decimal numbers, separated by commas, indicating the type of program interruption to be handled by the user's exit routine. Interruption types not specified are handled by the control program. The interruption types can be designated in any order as follows:

- One or more single numbers, each indicating the corresponding program interruption type.

- One or more pairs of decimal numbers, each pair indicating a range of corresponding interruption types. The second number must be higher than the first and the pair of numbers must be separated from each other by commas and enclosed in an additional set of parentheses.

For example, (4,8) indicates interruption types 4 and 8; ((4,8)) indicates interruption types 4 through 8. If a specified program interruption type is maskable, the corresponding bit is set to 1. The interruption types are as follows:

Number	Interruption Type
1	Operation
2	Privileged operation
3	Execute
4	Protection
5	Addressing

Number	Interruption Type (Cont'd)
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

The user-provided SPIE exit routine is executed whenever one of the types of specified interruptions occurs. The exit routine must be in virtual storage when it is required. Since the routine operates as a subroutine of the control program, it must return control to the control program.

Input to the SPIE exit routine is as follows.

Register 0: Control program information.

Register 1: Address of the PIE for the task that caused the interruption (Figure 57).

Register 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 PIE after control is returned but does not restore the contents of registers 3-13.

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

The exit routine can alter the contents of the registers that are to be returned to the interrupted program. For registers 3-13, the routine alters the contents of the actual registers. For registers 14, 15, 0, 1, and 2, the routine alters the contents of the register save area in the PIE. This is because the control program reloads these registers from this save area when it returns control to the interrupted program.

The exit routine can also alter the last four bytes of the OPSW in the PIE. By changing the OPSW, the routine can select any return point in the interrupted program.

The control program returns control to the interrupted program by loading a PSW constructed from the possibly modified OPSW saved in the PIE.

If a program interruption occurs when the program interruption exit routine is in control, the control program exit routine is given control.

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 description of the standard form of SPIE explains the function of each operand. The description of the standard form also indicates which operands are always optional and which are required in at least one of the pair of list and execute forms. The format description below indicates the optional and required operands in the list form only.

[symbol]	SPIE	[interruption exit address][,(interruptions)],MF=L

symbol
: is any symbol valid in the assembler language.

address
: is any address that may be written in an A-type address constant.

interruptions
: are one or more decimal digits separated by commas.

MF=L
: indicates the list form of the SPIE macro instruction.

[ ] indicates optional name or operands.

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 interruption 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 operand.

The address of the remote control program parameter list associated with any previous SPIE environment is returned by the SPIE macro instruction.

The description of the standard form of SPIE explains the function of each operand. The description of the standard form also indicates which operands are always optional and which are required in at least one of the pair of list and execute forms. The format description below indicates the optional and required operands in the execute form only. If the address of a previous PICA is used, only the MF operand should be coded.

```
┌──────────┬──────┬──────────────────────────────────────────────────┐
│ [symbol] │ SPIE │ [interruption exit address][,(interruptions)]    │
│          │      │ ,MF=(E,{control program list address})           │
│          │      │        {          (1)                }           │
└──────────┴──────┴──────────────────────────────────────────────────┘
```

symbol
: is any symbol valid in the assembler language.

address
: is any address that is valid in an RX-type instruction, or one of general registers 2 through 12, previously loaded with the indicated address. You may designate the register symbolically or with an absolute expression; always code it within parentheses.

interruptions
: are one or more decimal numbers separated by commas.

```
MF=(E,{control program list address})
      {          (1)                }
```
indicates the execute form of the macro instruction using a remote control program parameter list (program interruption control area). The address of the control program parameter list can be coded as described under "address," or can be loaded into register 1, in which case code MF=(E,(1)).

[ ] indicates optional name or operand; select one from vertical stack within { }.

STATUS -- Change Subtask Status (VS2 only)

The STATUS macro instruction lets the programmer change the dispatchability status of one or all of his program's subtasks. One use of the STATUS macro instruction is to restart subtasks that were stopped when an attention exit routine was entered.

The STATUS macro instruction is used only in a VS2 environment. It is ignored when it is issued in VS1.

The STATUS macro instruction is written as follows:

```
┌────────┬──────┬──────────────────────────────────────────────────────┐
│[symbol]│STATUS│{START}[,TCB=subtask tcb address]                     │
│        │      │{STOP }                                               │
└────────┴──────┴──────────────────────────────────────────────────────┘
```

START
: indicates that the STOP/START count in the task control block specified in the TCB operand will be decreased by 1. If the TCB operand is not coded, the STOP/START count is decreased by 1 in subtask task control blocks for all the subtasks of the originating task.

STOP
: indicates that the STOP/START count in the task control block specified in the TCB operand will be increased by 1. If the TCB operand is not coded, the STOP/START count is increased by 1 in the task control blocks for all the subtasks of the originating task.

TCB=
: is the address of a task control block that is to have its STOP/START count adjusted. If this operand is not specified, the STOP/START count is adjusted in the task control blocks for all the subtasks of the originating task.

Control is returned to the instruction following the STATUS macro instruction. When control is returned, register 15 contains one of the following return codes:

Hexadecimal Code	Meaning
00	Successful.
04	The specified task control block does not belong to a subtask of the originating task. The STATUS macro intruction was ignored.

STIMER -- Set Interval Timer

Use the STIMER macro instruction to set a programmed timer to a specified time interval (less than 24 hours) or to an interval that will expire at a specified time of day. The interval is decreased continuously. 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 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) or a "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 or suspend 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 shown in the following format description. The operand combinations in the shaded area of the format description may only be used in a VS2 system.

```
┌────────┬───────┬──────────────────────────────────────────┬─────────────────┐
│[symbol]│STIMER │(REAL[,timer completion exit address])  (,DINTVL=address │
│        │       │{TASK[,timer completion exit address]}  {,BINTVL=address │
│        │       │(WAIT                                )  {,TUINTVL=address│
│        │       │                                        {,TOD=address    │
│        │       │                                        (,MIC=address    │
└────────┴───────┴──────────────────────────────────────────┴─────────────────┘
```

REAL
    is written as shown. It specifies that the timer interval is a real-time interval and is to be decreased continuously. If the TOD operand is coded, the interval expires at the indicated time of day. You can also specify a real-time interval by using the WAIT operand.

TASK
    is written as shown. It specifies that the timer interval is a task time interval and is to be decreased only when the associated task is active.

WAIT
    is written as shown. It specifies that the time interval is a real-time interval and is to be decreased continuously. The job step is to be placed in the wait condition until the interval expires.

timer completion exit address
    is the address of the timer completion exit routine to be given control after the specified time interval expires. The exit routine is given control by means of an interruption of the task that was active when the STIMER macro instruction was issued; the routine must be in virtual storage when it is required. The contents

----
[ ] indicates optional name or operand; select one from vertical stack within { }.

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.

DINTVL=
: is the address in virtual storage of a doubleword on a doubleword boundary 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; and
h is hundredths of a second (maximum value 9).

BINTVL=
: is the address in virtual storage of 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 0.01 second.

TUINTVL=
: is the address of 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 (26.04166 microseconds).

TOD=
: is the address of 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. If TASK is specified, the time of day is interpreted as though the DINTVL operand had been specified.

MIC=
: is the address of 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 digit of the interval value.

Notes:

- The time interval specified by an STIMER macro instruction has no relation to the time interval specified in an EXEC statement.
- If issued by a timer completion exit routine, an STIMER macro instruction acts as a NOP instruction. However, the STIMER issued from a timer completion exit routine must not specify the same exit routine or an infinite loop results.

- 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.

When you are using VS, 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.

TIME -- Provide Time and Date

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

The date is returned in register 1 as packed decimal digits of the form 00 YY DD DC, 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 twenty-four-hour clock, returned in the form designated by the operand shown below. For the DEC, BIN, and TU operands, the time of day is returned in register 0. For the MIC, address operand, the time of day is returned in the specified address, and register 0 is set to zero. If the operand is omitted, DEC is assumed.

The TIME macro instruction is written as follows:

```
┌───────────┬───────┬─────────────────┐
│ [symbol]  │ TIME  │ ┌DEC          ┐ │
│           │       │ │BIN          │ │
│           │       │ │TU           │ │
│           │       │ └MIC,address  ┘ │
└───────────┴───────┴─────────────────┘
```

DEC
: is written as shown. 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; and
h is hundredths of second (maximum value 9).

BIN
: is written as shown. 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.

TU
: is written as shown. Time of day is returned in register 0 as an unsigned 32-bit binary number. The low-order bit is equivalent to 26.04166 microseconds (one timer unit).

MIC
: is written as shown. It requests the time of day in microseconds.

[ ] indicates optional name or operand; select one or none from within [ ]; ____ indicates an assumed value.

address
: is the address of an 8-byte area in storage where the time of day is returned in microseconds with bit 51 equivalent to one microsecond.

If the MIC,address operand is specified, register 15 contains one of the following return codes when control is returned to the user:

Hexadecimal Code	Meaning
00	Successful.
04	Unsuccessful. The specified address is not valid. The date is stored in register 1; register 0 contains 0.

TTIMER -- Test Interval Timer

In VS1, or in VS2 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 (26 microsecond units) remaining in the interval. If a time interval has not been set, register 0 contains 0. TTIMER can also be used to cancel the remaining time interval.

If MIC is specified in a VS2 system, the remaining time is returned to the doubleword area specified in the address. Bit 51 of the area is the low-order digit of the interval value. If a time interval has not been set the area is set to 0.

The operand combinations in the shaded area of the format description may only be used in a VS2 system.

The TTIMER macro instruction is written as follows:

[symbol]	TTIMER	[CANCEL]	$\left[\begin{Bmatrix}\underline{\text{TU}}\\ \text{MIC,address}\end{Bmatrix}\right]$

CANCEL
is written as shown. It indicates that the remaining time interval and exit routine, if any, are to be canceled. If WAIT was coded in the STIMER macro instruction that established the interval, the task is <u>not</u> taken out of the wait condition. If CANCEL is not designated, the unexpired portion of the time interval remains in effect.

TU
is written as shown. Remaining time in the interval is returned in register 0 as an unsigned 32-bit binary number. The low-order bit is equivalent to 26.04166 microseconds (one timer unit).

MIC
is written as shown. It requests the remaining time in the interval to be retuned in microseconds. Address is the doubleword area on a doubleword boundary where the remaining interval is to be stored.

Note: For further information about the use of TTIMER, refer to the description of the STIMER macro instruction.

If MIC, address is specified, register 15 contains one of the following return codes when control is returned to the user:

Hexadecimal Code	Meaning
00	The area specified by address contains the time remaining in the interval.
04	The area specified is not contained within the requestor's allocated storage. If cancel was coded, the interval was not canceled.

[ ] indicates optional name or operand; ____ indicates an assumed value.

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 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:

```
┌──────────┬──────┬──────────────────────────────────────┐
│ [symbol] │ WAIT │ [number of events,]{ECB=address    } │
│          │      │                    {ECBLIST=address} │
└──────────┴──────┴──────────────────────────────────────┘
```

number of events
: is a decimal integer from 0 to 255. Zero is an effective NOP instruction; one is assumed if the operand is omitted. The number of events must not exceed the number of event control blocks.

ECB=
: is the address of the event control block representing the single event that must occur before processing can continue. The operand is valid only if the number of events is specified as one or is omitted.

ECBLIST=
: is the address of a virtual storage area containing one or more consecutive fullwords on a fullword boundary. Each fullword contains the address of an event control block; the high-order bit in the last word (address) must be set to 1 to indicate the end of the list. The number of event control blocks must be equal to or greater than the specified number of events.

Caution: A job step with all of its tasks in a WAIT condition is terminated upon expiration of the time limits that apply to it.

Example: You have previously initiated one or more activities to be completed asynchronously to your processing. As each activity was initiated, you set up an ECB in which bits 0 and 1 were set to 0. You now wish to suspend your task via the WAIT macro instruction until a specified number of these activities has been completed.

[ ] indicates optional name or operand; select one from vertical stack within { }

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.

Event Control Block

The event control block is used for communication between various components of the control program, as well as between processing programs and the control program. An ECB is the subject of WAIT and POST macro instructions. Figure 58 shows the format of the event control block. A description of its fields follows the illustration.

```
┌──────────────────────────────────────────────────────────────────┐
│+0                +1                +2                +3          │
│                                                                  │
│W│C│              │                 │                 │           │
└─┴─┴──────────────┴─────────────────┴─────────────────┴───────────┘
```

Figure 58. Event control block

Displacement	Bytes and Alignment	Field Name	Hex. Dig.	Field Description, Contents, Meaning
+0	1			Awaiting completion of an event:
		1...		W - Waiting for completion of an event.
				After completion of an event:
		.1..		C - The event has completed.
		..xx xxxx		Completion code. One of the following completion codes will appear at the completion of a channel program:
				Access Methods Except BTAM and TCAM.
			7F	Channel program has terminated without error. (CSW contents useful.)
			41	Channel program has terminated with permanent error. (CSW contents useful.)
			42	Channel program has terminated because a direct access extent address has been violated. (CSW contents do not apply.)
			44	Channel program has been intercepted because of permanent error associated with device end for previous request. You may reissue the intercepted request. (CSW contents do not apply.)
			48	Request element for channel program has been made available after it has been purged. (CSW contents do not apply.)

Displacement	Bytes and Alignment	Field Name	Hex. Dig.	Field Description, Contents, Meaning
			4F	Error recovery routines have been entered because of direct access error but are unable to read home address or record 0. (CSW contents do not apply.)
				BTAM
			7F	Normal completion.
			41	Completed with an I/O error.
			48	Enable command halted, or, I/O operation purged.
				TCAM
			7F	Normal completion (work unit in work area).
			70	The SETEOF macro instruction was issued in the message command program (no work unit in work area).
			50	Message was not found when the READ macro instruction was issued in conjunction with the POINT macro instruction to retrieve a message.
			5C	Congested destination message queue data set (write only).
			58	Sequence error.
			54	Invalid message destination.
			52	Work area overflow.
			02	End-of-queue condition (not end-of-file).
			01	Read-ahead queue empty, but destination queue not empty.
			40	Data is on read-ahead queue.
+1	3			Awaiting completion of an event: Request block address. After completion of the event: Zeros, or remainder of completion code.

WAITR -- Wait for One or More Events

The WAITR macro instruction is coded and is executed in exactly the same manner as the WAIT macro instruction.

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 shown below. Information about the list and execute forms follows this description.

[symbol]	WTL	'message'

message
    is the message to be written to the system log. The message must be enclosed in apostrophes, which will not appear in the log. The message is limited to 126 characters.

[ ] indicates optional name.

WTL -- List Form

Use the list form of the WTL macro instruction to construct a control program parameter list. The message operand must be provided in the list form of the macro instruction. The description of the standard form of the WTL macro provides the requirements for writing the message.

The list form of the WTL macro is written as follows:

[symbol]	WTL	'message',MF=L

message
    is any character string valid in a C-type (character) DC instruction.

MF=L
    indicates the list form of the WTL macro instruction.

[ ] indicates optional name.

WTL -- Execute Form

A remote control program parameter list is used in the execute form of the WTL macro instruction. 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 is written as follows:

```
┌──────────┬─────┬────────────────────────────────────────┐
│ [symbol] │ WTL │ MF=(E,{control program list address})  │
│          │     │       {          (1)                 } │
└──────────┴─────┴────────────────────────────────────────┘
```

```
MF=(E,{control program list address})
      {         (1)                 }
```

indicates the execute form of the macro instruction using a remote control program parameter list. The address of the control program parameter list can be loaded into register 1, in which case code MF=(E,(1)). If the address is not loaded into register 1, code it as any address that is valid in an RX-type instruction, or one of the general registers 2-12, previously loaded with the address. You can designate a register symbolically or with an absolute expression; always code it within parentheses.

[ ] indicates optional name; select one from vertical stack within { }.

WTO -- Write to Operator (VS1 Without Multiple Console Support)

The WTO macro instruction causes a message to be written to the operator's console.

The standard form of the WTO macro instruction is written as shown below. The operands in the shaded area of the format description are used in operating systems that include the Multiple Console Support (MCS) option; they are ignored if coded in an operating system that does not include the MCS option, except for descriptor codes 1 and 2, and routing code 11 which designates a Write-to-Programmer request. Routing and descriptor codes are described in Appendix C.

If you code a WTO macro instruction with a routing code of 11 in an operating system without MCS, this message will go to the system message class data set and not to the operator's console. If you want the message to also appear on the operator's console, code the appropriate routing code (as described in Appendix C) in addition to routing code 11. For example:

WTO 'message', ROUTCDE=11	Results in a Write-to-Programmer message. The message will appear only on the system message class data set.
WTO 'message', ROUTCDE=(x,11)	Where x represents any valid routing code other than 11 (see Appendix C for a description of these codes). Results in both a Write-to-Programmer and a Write-to-Operator message. The message will appear on both the system message class data set and on the operator's console.

The operands in the nonshaded area can be coded with any configuration of the operating system. Information about the list and execute forms follows the write-up for WTO with MCS support.

[symbol]	WTO	'message'[,ROUTCDE=(number[,number...]) [,DESC=number]

message
: is 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 maximum message length is 120 characters (bytes) for a user non-action message. All other messages may be as long as 121 bytes. The message is assembled as a variable-length record.

[ ] indicates optional name and operands.

WTO -- Write to Operator (VS1 With Multiple Console Support)

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 shown below. Information about the list and execute forms follows this description.

```
┌───────────┬──────┬──────────────────────────────────────────┐
│ [symbol]  │ WTO  │{'message'                      }          │
│           │      │{('text'[,line type]),...      }          │
│           │      │ [,ROUTCDE=(number[,number],...)]          │
│           │      │ [,DESC=number]                            │
└───────────┴──────┴──────────────────────────────────────────┘
```

message
: is the message to be written to one or more operator consoles. The message must be enclosed in apostrophes; the apostrophes do 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 maximum message length is 120 characters (bytes) for a user non-action message. All other messages may be 121 characters. The message is assembled as a variable-length record.

Note: All WTO messages with a descriptor code 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.

('text'[,line type])
: is used to write a multiple-line message to the operator. The message may be up to ten lines long (if more than ten lines are coded in the macro, the macro is not generated and an MNOTE is issued). This limit does not include the control line (message IEE932I), see item C below.

text
: is one line of the multiple-line message to be passed to the operator. A line consists of a character string enclosed in apostrophes (the apostrophes do not appear on the operator's 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 (see Figure 59).

line type
: is an alphabetic indicator defining 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).

[ ] indicates optional name and operands.

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=
: specifies the routing codes to be assigned to the message. "Number" must be a routing code from 1 through 16. Routing codes are defined in Appendix C. If the ROUTCDE operand is omitted but DESC is specified, routing code 2 is assigned.

DESC=
: specifies the message descriptor code(s) to be assigned to the message. "Number" must be a descriptor code from 1 through 16. Descriptor codes are defined in Appendix C. If the DESC operand is omitted, no descriptor code is assigned.

If both the ROUTCDE and DESC parameters are omitted, no routing or descriptor codes are assigned.

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

Note: The two operands available to the system programmer are MSGTYP and MCSFLAG. They are discussed in Appendix C.

Line Type	VS1	VS2
C	31 characters	34 characters
L	71 characters	70 characters
D	71 characters	70 characters
DE	71 characters	70 characters
Note: L, D, and DE lines displayed on a 2250 display console will be truncated to 70 characters.		

Figure 59. Maximum 'text' field characters in a multiple-line WTO 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.
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.
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 -- Write to Operator (VS2 With Multiple Console Support)

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:

```
┌─────────┬─────────┬──────────────────────────────────────────────────┐
│[symbol] │  WTO    │ ⎧'message'                    ⎫                  │
│         │         │ ⎩('text'[,line type]),...     ⎭                  │
│         │         │ [,ROUTCDE=(number[,number],...)]                 │
│         │         │ [,DESC=(number[,number],...)]                    │
│         │         │ [,AREAID=char]                                   │
└─────────┴─────────┴──────────────────────────────────────────────────┘
```

message
: is the message to be written to one or more operator consoles. The message must be enclosed in apostrophes (the apostrophes do 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 maximum message length is 124 characters (bytes). The message is assembled as a variable-length record.

Note: All WTO messages with a descriptor code of 1 or 2 are action messages. An indicator is printed before the first character of an action message to indicate a need for operator action, but this does not reduce the maximum length of an action message.

('text'[,line type])
: is used to write a multiple-line message to the operator. The message may be up to ten lines (if more than ten lines are passed by a program, the system truncates the message at the end of the tenth line). This limit does not inlude the control line (message IEE932I).

text
: is one line of the multiple-line message to be passed to the operator. A line consists of a character string enclosed in apostrophes (the apostrophes do not appear on the operator's 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 (see Figure 59).

line type
: is an alphabetic indicator defining 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).

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 mes-

sage 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' patameter 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=
specifies the routing codes to be assigned to the message. Number must be a routing code from 1 through 16. (Routing codes are defined in Appendix C). If the ROUTCDE operand is omitted but the DESC is specified, routing code 2 is assigned.

DESC=
specifies the message descriptor code or codes to be assigned to the message. Number must be a descriptor code from 1 through 16. (Descriptor codes are defined in Appendix C.) If the DESC operand is omitted, no description code is assigned.

AREAID=
specifies a display area of the console screen on which a multiple line message is to be written. "char" may be any alphabetic character A-Z.

Z designates the message area (the screen's general message area, rather than a defined display area); it is assumed nothing is specified.

The AREAID parameter is only useful for out-of-line (descriptor code 8 and 9) MLWTO messages which are to be sent to CRT consoles.

If both the ROUTCDE and DESC parameters are omitted, the routing code specified in the OLDWTOR operand of the system generation SCHEDULR macro instruction is assigned. If the OLDWTOR operand is omitted, no routing code is assigned.

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

Note: The two operands available to the system programmer are MSGTYP and MCSFLAG. They are discussed in Supervisor Services and Macro Instructions for the System Programmer.

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.
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.
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

Use the list form of the WTO macro instruction to construct a control program parameter list. The message operand must be provided in the list form of the macro. The description of the standard form of the WTO macro provides the requirements for writing the message.

The format description below indicates the optional and required operands for the list form. The operands in the shaded area of the format description are used with the Multiple Console Support (MCS) option; they are ignored if coded without MCS, except routing codes 1 and 2 which designate a Write-to-Master Console and routing code 11 which designates a Write-to-Programmer request. (See the standard form of the WTO macro for a description of this exception.)

```
┌───────────┬─────┬─────────────────────────────────────────┐
│ [symbol]  │ WTO │{('text'[,line type]),...}                │
│           │     │{'message'               }                │
│           │     │ [,ROUTCDE=(number[,number...])           │
│           │     │ [,DESC=number],MF=L                      │
└───────────┴─────┴─────────────────────────────────────────┘
```

message
: is a character string valid in a character (C-type) DC instruction, except the line control character (punch combination 11-9-5).

'text'
: is a character string valid in a C-type DC instruction except for the New Line control character.

line type
: is an alphabetic symbol indicating the type of information contained in the 'text' parameter.

MF=L
: indicates the list form of the WTO macro.

ROUTCDE=
: specifies the routing codes to be assigned to the message.

DESC=
: specifies the message descriptor code to be assigned to the message.

Note: Two additional operands available to the system programmer (MSGTYP and MCSFLAG) are discussed in Appendix C.

[ ] indicates optional name or operand.

WTO -- Execute Form

A remote control program parameter list is used in the execute form of the WTO macro instruction. The parameter list can be generated by the list form of WTO. The message cannot be modified in the execute form of the macro.

The execute form of the WTO macro is written as follows:

```
┌───────────┬─────┬────────────────────────────────────────────┐
│ [symbol]  │ WTO │ MF=(E,{control program list address})      │
│           │     │       {            (1)                }    │
└───────────┴─────┴────────────────────────────────────────────┘
```

```
MF=(E,{control program list address})
      {            (1)               }
```

indicates the execute form of the macro instruction using a remote control program parameter list. If you have loaded the address of the control program parameter list into register 1, code MF=(E,(1)). If the address is not loaded into register 1, code it as any address that is valid in an RX-type instruction, or one of the general registers 2-12, previously loaded with the address. Designate the register symbolically or with an absolute expression; always code it within parentheses.

[ ] indicates optional name; select one from vertical stack within { }.

WTOR -- Write to Operator With Reply (Without Multiple Console Support)

The WTOR macro instruction causes a message requiring a reply to be written to the operator's console, and 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 shown below. The operands in the shaded area of the format description are used in an operating system that includes the Multiple Console Support (MCS) option; they are ignored if coded in an operating system that does not include the MCS option, except for descriptor codes 1 and 2, and routing code 11 which designates a Write-to-Programmer request. If a WTOR message is coded with a routing code of 11 in an operating system that does not include the MCS option, the message portion of the message will go to both the system message class data set and the operator's console. The operands in the nonshaded area can be coded with any configuration of the operating system. Information about the list and execute forms follows this description.

[symbol]	WTOR	'message',reply address,length of reply, ecb address[,ROUTCDE=(number[,number,...])] [,DESC=number]

message
: is the message to be written to the operator's console. The message must be enclosed in apostrophes(the apostrophes do 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 maximum message length is 117 characters (bytes). The message is assembled as a variable-length record. No requirement exists to pad the message with blanks.

    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 address
: is 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.

length of reply
: is the length, in bytes, of the reply message. The maximum reply length is 119 bytes. The minimum reply length is one byte.

ecb address
: is the address of the event control block to be used by the control program to indicate the completion of the reply.

[ ] indicates optional name or operand.

WTOR -- Write to Operator With Reply (With Multiple Console Support)

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

The standard form of the WTOR macro is written as follows. Information about the list and execute forms follows this description.

[symbol]	WTOR	'message',reply address,length of reply, ecb address[,ROUTCDE=(number[,number...]) [,DESC=number]

message
: is 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 maximum message length is 117 characters (bytes) in VS1, 121 characters in VS2. The message is assembled as a variable-length record. No requirement exists to pad the message with blanks.

    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 address
: is 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.

length of reply
: is the length, in bytes, of the reply message. The maximum reply length is 115 characters when the operator enters REPLY id, 'reply' and 119 characters when the operator enters R id, 'reply'. The minimum reply length is one byte.

ecb address
: is the address of the event control block to be used by the control program to indicate the completion of the reply.

ROUTCDE=
: specifies the routing codes to be assigned to the message. Number must be a routing code from 1 through 16. Routing codes are defined in Appendix C. If the ROUTCDE operand is omitted but the DESC operand is specified, routing code 2 is assigned.

DESC=
: specifies the message descriptor code(s) to be assigned to the message. Number must be a descriptor code from 1 through 16. Descriptor codes are defined in Appendix C. If the DESC operand is omitted, no descriptor code is assigned.

If both the ROUTCDE and DESC operands are omitted, no routing or descriptor codes are assigned.

[ ] indicates optional name or operand.

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

Note: The two operands available to the system programmer are MSGTYP and MCSFLAG. They are discussed in Appendix C (VS1) and in VS2 Planning and Use Guide.

WTOR -- List Form

Use the list form of the WTOR macro instruction to construct a control program parameter list. The message operand must be provided in the list form.

The description of the standard form of the WTOR macro provides the requirements for writing the message and the explanation of the function of each operand. The description of the standard form also indicates which operands are totally optional and which are required in at least one of the pair of list and execute forms. The format description below indicates the optional and required operands in the list form only. The operands in the shaded area of the format descriptions are used with the Multiple Console Support (MCS) option; they are ignored if coded without MCS, except routing codes 1 and 2 which designate a Write-to-Master-Console and code 11 in the ROUTCDE operand which designates a Write-to-Programmer request. (See the standard form of the WTOR macro instruction for a description of this exception.)

The list form of the WTOR macro is written as follows:

[symbol]	WTOR	'message',[reply address],[length of reply] ,[ecb address] [,ROUTCDE=(number[,number...]) [,DESC=number] ,MF=L

symbol
    is any symbol valid in the assembler language.

address
    is any address that can be written in an A-type address constant.

length
    is any absolute expression valid in the assembler language.

message
    is a character string valid in a character (C-type) DC instruction except the line control character (punch combination 11-9-5).

ROUTCDE=
    specifies the routing codes to be assigned to the message.

DESC=
    specifies the message descriptor code to be assigned to the message.

MF=L
    indicates the list form of the WTOR macro.

Note: Two additional operands (MSGTYP and MCSFLAG) available to the system programmer are discussed in Appendix C (VS1) and VS2 Planning and Use Guide.

[ ] indicates optional name or operand.

WTOR -- Execute Form

A remote control program parameter list is used in the execute form of the WTOR macro instruction. The parameter list can be generated by the list form of WTOR.

The description of the standard form of the WTOR macro provides the explanation of the function of each operand. The description of the standard form also indicates which operands are totally optional and which are required in at least one of the pair of list and execute forms. The format description below indicates the optional and required operands in the list form only. The comma before the first operand is required to indicate the absence of the message operand, which is not allowed in the execute form.

The execute form of the WTOR macro is written as follows:

```
┌───────────┬───────┬─────────────────────────────────────────────────────┐
│ [symbol]  │ WTOR  │ ,[reply address],[length of reply],[ecb address]    │
│           │       │ ,MF=(E,{control program list address})              │
│           │       │        {          (1)                }              │
└───────────┴───────┴─────────────────────────────────────────────────────┘
```

symbol
    is any symbol valid in the assembler language.

address
    is any address that is valid in an RX-type instruction, or one of general registers 2 through 12, previously loaded with the indicated address. Designate the register symbolically or with an absolute expression; always code it within parentheses.

length
    is any absolute expression that is valid in the assembler language, or one of general registers 2 through 12, previously loaded with the indicated value. Designate the register symbolically or with an absolute expression; always code it within parentheses.

```
MF=(E,{control program list address})
      {          (1)                }
```
    indicates the execute form of the macro instruction using a remote control program parameter list. The address of the control program parameter list can be coded as described under "address," or can be loaded into register 1, in which case code MF=(E,(1)). The parameter list must be aligned on a fullword boundary. The list form of WTOR provides this alignment.

[ ] indicates optional name or operand; select one from vertical stack within { }.

XCTL -- Pass Control to a Program in Another Load Module

The XCTL macro instruction causes control to be passed to a specified entry point in another load module; the entry point name must be a member name or an alias in a directory of a partitioned data set. The load module containing the entry point 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 if no other requirement exists for that load module. The program executing the XTCL 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 responsibility 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 point cannot be located, the task is abnormally terminated.

The standard form of the XCTL macro instruction is written as shown below. Information about the list and execute forms follows this description.

[symbol]	XCTL	[(reg1[,reg2])],{EP=symbol
		EPLOC=address of name
		DE=address of list entry}
		[,DCB=dcb address]

(reg1, reg2)
: is the range of registers from 2 through 12 to be restored from the save area pointed to by register 13. The value of the reg1 operand must be less than the value of the reg2 operand. If the reg2 operand is omitted, only the register specified is loaded; if both operands are omitted, the contents of the registers are not altered.

EP=
: is the entry point name in the program to be given control. The name must be padded with blanks on the right to eight bytes if necessary. If the specified entry point cannot be found, the task is abnormally terminated.

EPLOC=
: is the address of the entry point name described above. Pad the name with blanks to eight bytes, if necessary.

DE=
: is the address of the name field of a 58-byte list entry for the entry point name. The list entry is constructed using the BLDL macro instruction using a length specification of 58 bytes. The DCB operand must indicate the same data control block used in the BLDL macro instruction.

[ ] indicates optional name or operand; select one from vertical stack within { }.

DCB=

is the address of the data control block for the partitioned data set containing the entry point name described above. The DCB must not be defined in the program issuing the XCTL.

If the DCB operand is omitted or if DCB=0 is specified when the XCTL macro instruction is issued by the _job step task_, the data sets referred to by either the STEPLIB or JOBLIB DD statement are first searched for the entry point name. If the entry point name is not found, the link library is searched.

If the DCB operand is omitted or if DCB=0 is specified when the XCTL macro instruction is issued by a _subtask_, the data sets associated with one or more data control blocks referred to by 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 description of the standard form of XCTL explains the function of each operand. The description of the standard form also indicates which operands are always optional and which are required in at least one of the pair of list and execute forms. The format description below indicates the optional and required operands in the list form only.

```
┌───────────┬───────┬──────────────────────────────────────────────────┐
│ [symbol]  │ XCTL  │ ┌EP=symbol                  ┐[,DCB=dcb address] │
│           │       │ │EPLOC=address of name      │                   │
│           │       │ └DE=address of list entry   ┘                   │
│           │       │ ,SF=L                                            │
└───────────┴───────┴──────────────────────────────────────────────────┘
```

symbol
    is any symbol valid in the assembler language.

address
    is any address that may be written in an A-type address constant.

SF=L
    indicates the list form of the XCTL macro.

[ ] indicates optional name or operands; select one or none from vertical stack within [ ].

XCTL -- Execute Form

Two parameter lists are used in the XCTL macro instruction; 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 XCTL. If only the problem program parameter list is remote, operands 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, including the reg1,reg2 operand, can be specified

The description of the standard form of XCTL explains the function of each operand. The description of the standard form also indicates which operands are always optional and which are required in at least one of the pair of list and execute forms. The format description below indicates the optional and required operands in the execute form only.

```
┌───────────┬───────┬──────────────────────────────────────────────────┐
│ [symbol]  │ XCTL  │ [(reg1[,reg2])]                                  │
│           │       │ ┌,EP=symbol             ┐                        │
│           │       │ │,EPLOC=address of name │[,DCB=dcb address]      │
│           │       │ └,DE=address of list entry┘                      │
│           │       │ ⎧,MF=(E,⎧problem program list address⎫)⎫         │
│           │       │ ⎪       ⎩            (1)             ⎭)⎪         │
│           │       │ ⎨,SF=(E,⎧control program list address⎫)⎬         │
│           │       │ ⎪       ⎩            (15)            ⎭)⎪         │
│           │       │ ⎪,MF=(E,⎧address⎫),SF=(E,⎧address⎫)   ⎪         │
│           │       │ ⎩       ⎩  (1)  ⎭        ⎩ (15)  ⎭    ⎭         │
└───────────┴───────┴──────────────────────────────────────────────────┘
```

symbol
    is any symbol valid in the assembler language.

address
    is any address that is valid in an RX-type instruction, or one of general registers 2 through 12, previously loaded with the indicated address. Designate the register symbolically or with an absolute expression; always code it within parentheses.

MF=(E,{problem program list address | (1)})
    indicates the execute form of the macro instruction using a <u>remote problem program parameter list</u>. Any control program parameters specified are provided in a control program parameter list expanded in line. The address of the problem program parameter list can be coded as described under "address," or can be loaded into register 1, in which case code MF=(E,(1)).

SF=(E,{control program list address | (15)})
    indicates the execute form of the macro instruction using a <u>remote control program parameter list</u>. No problem program parameters can be specified. The address of the control program parameter list can be coded as described under "address," or can be loaded into register 15, in which case code SF=(E,(15)).

[ ] indicates optional name or operand; select one from vertical stack within { }; select one or none from vertical stack within [ ].

```
MF=(E,{address}),SF=(E,{address})
      { (1)   }         { (15)  }
```

indicates the execute form of the macro instruction using <u>both</u> a remote problem program parameter list and a remote control program parameter list. The addresses of the parameter lists are coded or loaded into registers 1 and 15, as explained above.

APPENDIX A: SUMMARY OF OPERANDS

Appendix A indicates how each operand may be coded in the standard and, where applicable, in the list and execute forms of each macro instruction. For example, in ATTACH macro instruction the DCB operand may be coded in the standard (S) form using registers 2-12 or as an A-type address constant, in the list (L) form as an A-type address constant, and in the execute (E) form using registers 2-12 or as an RX-type address constant. Only the indicated methods of coding should be used.

ABBREVIATIONS USED IN APPENDIX A

Abbreviation		Meaning
Sym		Any symbol valid in the assembler language.
Dec Dig		Any decimal digits, up to the value indicated in the associated macro instruction description. If both Sym and Dec Dig are checked, an absolute expression is also allowed.
Register		A general register, always coded within parentheses, as follows:
	(2-12) -	one of the general registers 2 through 12, previously loaded with the right-adjusted value or address indicated in the macro instruction description. The unused high-order bits must be set to zero. The register may be designated symbolically or with an absolute expression.
	(1) -	general register 1, previously loaded as indicated above. Designate the register as (1) only.
	(0) -	general register 0, previously loaded as indicated above. Designate the register as (0) only.
RX-type		Any address that is valid in an RX-type instruction (for example, LA) may be designated.
A-type		Any address that may be written in an A-type address constant may be designated.

Macro Instruction	Operands	Written as						
				Register				
		Sym	Dec Dig	(2-12)	(1)	(0)	RX-type	A-type adcon type
ABEND	completion code	S	S	S	S			
	DUMP	written as shown						
	STEP	written as shown						
ATTACH	ASYNCH=	YES or NO						
	DCB=			S E			E	S L
	DE=			S E			E	S L
	DPMOD=	S L E	S L E	S E				
	ECB=			S E			E	S L
	EP=	S L E						
	EPLOC=			S E			E	S L
	ETXR=			S E			E	S L
	FPREGSA=	YES or NO						
	GSPL=			S E			E	S L
	GSPV=	S L E	S L E	S E				
	LPMOD=	S L E	S L E	S E				
	PARM=			S E			E	S
	PURGE=	QUIESCE, HALT, or NONE						
	SHSPL=			S E			E	S L
	SHSPV=	S L E	S L E	S E				
	STAI=			S E			E	S L
	SZERO=	YES or NO						
	TASKLIB=			S E			E	S L
	TQE=	YES or NO						
	VL=1	written as shown						
CALL	entry point name	S E						
	address parameters			S E			E	S L
	VL	written as shown						
	ID=	S E	S E					
CHAP	priority change value	S	S	S		S		
	tcb location address			S	S		S	
DELETE	DE=			S		S	S	
	EP=	S						
	EPLOC=			S		S	S	

cro struction	Operands	Written as						
		Sym	Dec Dig	Register			RX-type	A-type adcon type
				(2-12)	(1)	(0)		
:Q	qname address			S E			E	S L
	rname address			S E			E	S L
	rname length	S L E	S L E	S E				
	STEP or SYSTEM	written as shown						
	RET=HAVE	written as shown						
:TACH	tcb location address	S		S	S		S	
	STAE=	YES or NO						
:M	MSG=			S	S			
	MSGLIST=	S		S	S		S	
:R	reg1	S	S					
	reg2	S	S					
IQ	qname address			S E			E	S L
	rname address			S E			E	S L
	E or S	written as shown						
	rname length	S L E	S L E	S E				
	STEP or SYSTEM	written as shown						
	RET=	TEST, USE, or HAVE						
:EEMAIN	E, L, R or V	written as shown						
	A=(with E, L, or V)			S E			E	S L
	A=(with R)			S	S		S	
	LA=			S E			E	S L
	LV=(with E)	S L E	S L E	S E				
	LV=(with R)	S	S	S		S		
	SP=(with E, L, or V)	S L E	S L E	S E				
	SP=(with R)	S	S	S		S		
ETMAIN	EC, EU, LC, LU, VC, or VU	refer to macro description						
	A=			S E			E	S L
	BNDRY=	DBLWD or PAGE						
	LA=			S E			E	S L
	LV=(with E)	S L E	S L E	S E				

		Written as						
				Register				
Macro Instruction	Operands	Sym	Dec Dig	(2-12)	(1)	(0)	RX-type	A-typ adcon type
GETMAIN (cont'd)	LV(with R)	S	S	S		S		
	SP=(with E, L, or V)	S L E	S L E	S E				
	SP=(with R)	S	S	S		S		
GTRACE	DATA=			S			S	S L E
	LNG=	S L E	S L E	S L E				
	FID=	S L E	S L E	S L E				
	ID=	S E	S E					
IDENTIFY	ENTRY=			S	S		S	
	EP=	S						
	EPLOC=			S		S	S	
LINK	DCB=			S E			E	S L
	DE=			S E			E	S L
	EP=	S L E						
	EPLOC=			S E			E	S L
	ID=	S E	S E					
	PARAM=			S E			E	S
	VL=1	written as shown						
LOAD	DCB=			S	S		S	
	DE=			S		S	S	
	EP=	S						
	EPLOC=			S		S	S	
PGRLSE	LA=			S E		S E		S L E
	HA=			S E	S E			S L E
	list addr=						E	
	reg 3=			E				
POST	ecb address			S	S		S	
	completion code	S	S	S		S		
RETURN	(reg1,reg2)		S					
	T	written as shown						
	RC=	S	S	or (15)				

acro nstruction	Operands	Written as						
				Register				
		Sym	Dec Dig	(2-12)	(1)	(0)	RX-type	A-type adcon type
AVE	(reg1,reg2)		S					
	T	written as shown						
	identifier name	character string or *						
EGWT	external segment name	S						
NAP	DCB=			S E			E	S L
	ID=	S L E	S L E	S E				
	LIST=			S E			E	S L
	PDATA	refer to macro description						
	SDATA	refer to macro description						
	STORAGE			S E			E	S L
	TCB=			S E			E	S
PIE	interruption exit address			S E			E	S L
	interruptions		S L E					
TATUS	STOP or START	written as shown						
	TCB=			S			S	
TIMER	REAL, TASK or WAIT	written as shown						
	timer completion exit addr			S		S	S	
	BINTVL=			S	S		S	
	DINTVL=			S	S		S	
	MIC=			S	S		S	
	TOD=			S	S		S	
	TUINTVL=			S	S		S	
IME	DEC or BIN or TU	written as shown						
	MIC	written as shown						
	address			S		S	S	
TIMER	CANCEL	written as shown						
	TU or MIC	written as shown						
AIT AITR	number of events	S	S	S		S		
	ECB=			S	S		S	
	ECBLIST=			S	S		S	
TL	message	any message within apostrophes						

Macro Instruction	Operands	Written as						
				Register				
		Sym	Dec Dig	(2-12)	(1)	(0)	RX-type	A-ty adco type
WTO	message	any message within apostrophes						
	ROUTCDE=		S L					
	DESC=		S L					
WTOR	message	any message within apostrophes						
	reply address			S E			E	S L
	length of reply	S L E	S L E	S E				
	ecb address			S E			E	S L
	ROUTCDE=		S L					
	DESC=		S L					
XCTL	(reg1,reg2)		S E				E	S
	DCB=			S E			E	S L
	DE=			S E			E	S L
	EP=	S L E						
	EPLOC=			S E			E	S
	HIARCHY=		S L E					
	PARAM=			E			E	

Explanation of style

Words in all capitals are coded as shown; appropriate values are to be substituted for words in lower case letters. Shaded operands may only be used in a VS2 system. Brackets, [ ], enclose operands that may be used or omitted as required; stacking within braces, { }, is used to indicate a choice of operands or values. Underlining, __, indicates a default value.

Footnotes:

*	In full-word on full-word boundary
**	In double-word on double-word boundary
+	Left justified in double-word on byte boundary
o	Multiple of eight; value given in bytes

LOAD MODULE CONTROL

```
Pass control      CALL       entry point name [,(address parameter [,address parameter]...)[,VL]]
and initiate
execution                    [,ID=0 to 65535]

Dynamically       LINK       ( EP=entry point name                   )
load and                     { EPLOC=address of entry point name+    } [,DCB=dcb address]
initiate                     ( DE=address of list entry              )
execution
                             [,PARAM=(address parameter [, address parameter]...) [,VL=1]]

                             [,ID=0 to 65535]

Transfer          XCTL                             ( EP=entry point name                  )
control                      [(register (s) 2-12)],{ EPLOC=address of entry point name+   }
                                                   ( DE=address of list entry             )

                             [,DCB=dcb address]

Dynamically       LOAD       ( EP= entry point name                  )
load                         { EPLOC=address of entry point name+    } [,DCB=dcb address]
                             ( DE=address of list entry              )

Delete            DELETE     ( EP=entry point name                   )
                             { EPLOC=address of entry point name+    }
                             ( DE=address of list entry              )

Identify          IDENTIFY   { EP=entry point name                   } ,ENTRY=entry point address
embedded                     { EPLOC=address of entry point name+    }
entry point

Load              SEGWT      external segment name
overlay
segment
```

SYNCHRONIZATION

```
Wait for          WAIT        [number of events,] { ECB=ecb address                           }
event                                             { ECBLIST=address of list of ecb addresses* }

Wait for event    WAITR       [number of events,] { ECB=ecb address                           }
while lower                                       { ECBLIST=address of list of ecb addresses* }
priority task
is executed

Post event        POST        ecb address [,0 to 16,777,215]
completion

Request           ENQ         (qname address,rname address,[E], [rname length] ,[SYSTEM],...)
control of                                                 [S]                  [STEP  ]
serially                      [,RET=TEST ]
reusable                      [,RET=USE  ]
resource                      [,RET=HAVE ]
                              [,RET=CHNG ]

Release           DEQ         (qname address,rname address, [rname length] ,[STEP  ],...)
serially                                                                     [SYSTEM]
reusable                      [,RET=HAVE]
resource
```

E means exclusive control S means shared control	default is E
SYSTEM	means resource used by more than one job
STEP	means resource used by issuing job

```
Set interval      STIMER      { REAL, [address of interval end routine] }
timer                         { TASK, [address of interval end routine] }
                              { WAIT                                    }

                              { ,DINTVL=address of decimal interval**                }
                              { ,BINTVL=address of binary interval in seconds*       }
                              { ,TUINTVL=address of binary interval in timer units*  }
                              { ,TOD=address of time-of-day of interval end**        }
                              { ,MIC=address                                         }

Test interval     TTIMER      [CANCEL] [, { TU          } ]
timer                                     { MIC,address }
```

TIME AND TIME INTERVALS FOR TIME, TTIMER, AND STIMER

Decimal (DEC and DINTVL operands):
Eight packed decimal digits in format HHMMSSth
- HH = hours in 24-hour clock
- MM = minutes
- SS - seconds
- t = tenths of seconds
- h = hundredths of seconds

Binary in seconds (BIN or BINTVL operands):
Unsigned 32-bit binary number in a full-word on full-word boundary; least significant bit has a value of 0.01 second

Binary in timer units (TU or TUINTVL operands):
Unsigned 32-bit binary number in a full-word on full-word boundary; least significant bit has a value of 1 timer unit (1 timer unit = 26 micro-seconds)

Binary in microseconds (MIC operand):
Unsigned 64-bit binary number in a double-word on a double-word boundary. Bit 51 is the low order digit of the interval value.

PROGRAM INTERRUPTION CONTROL

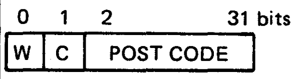
Enable and disable program interruptions and transfer control to interruption exit routine	SPIE	[interruption exit routine address] [,(interruption type [,interruption type] ...)]

INTERRUPTION TYPES FOR SPIE

Type	Meaning	Maskable	Type	Meaning	Maskable	Type	Meaning	Maskable
1	Operation	No	6	Specification	No	11	Decimal divide	No
2	Privileged operation	No	7	Data	No	12	Exponent overflow	No
3	Execute	No	8	Fixed-point overflow	Yes	13	Exponent underflow	Yes
4	Protection	No	9	Fixed-point divide	No	14	Significance	Yes
5	Addressing	No	10	Decimal overflow	Yes	15	Floating-point divide	No

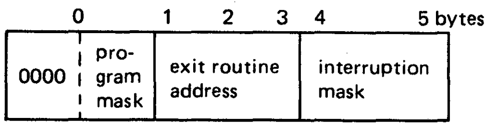
CONTROL BLOCKS

Event control block (ECB):

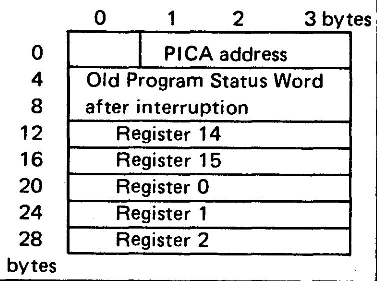


W = wait flag
C = completion flag

Program interruption control area (PICA):



Program interruption element (PIE):



GENERAL SERVICES

Delete message(s) from display	DOM	{ MSG=register containing 24-bit, right-justified message number MSGLIST=address of list of fullwords, each a 24-bit, right-justified identification number of message to be deleted }
Write to operator	WTO	'message' [,ROUTCDE=(number [,number] ,...)] { 'message' ('text' [,line type]), . . . } [DESC=message descriptor code(s)]
Write to operator and wait for reply	WTOR	'message',address of reply area,length of reply,ecb address [,ROUTCDE=(number [,number] ,...)] [,DESC=message descriptor code(s)]
Write to log	WTL	'message'
Divide extended precision floating point numbers	DXR	register containing dividend,register containing divisor Only registers 0 and 4 can be used; they may be specified in either order.
Get time and date	TIME	[<u>DEC</u> BIN TU MIC,address]

TIME AND TIME INTERVALS FOR TIME, TTIMER, AND STIMER

Decimal (DEC and DINTVL operands):
Eight packed decimal digits in format HHMMSSth
- HH = hours in 24-hour clock
- MM = minutes
- SS = seconds
- t = tenths of seconds
- h = hundredths of seconds

Binary in seconds (BIN or BINTVL operands):
Unsigned 32-bit binary number in a full-word on full-word boundary; least significant bit has a value of 0.01 second

Binary in timer units (TU or TUINTVL operands):
Unsigned 32-bit binary number in a full-word on full-word boundary; least significant bit has a value of 1 timer unit (1 timer unit = 26 micro-seconds)

Binary in microseconds (MIC operand):
Unsigned 64-bit binary number in a double-word on a double-word boundary. Bit 51 is the low order digit of the interval value.

Save register contents	SAVE	(register(s) 14 through 12) [,T] [,identifier] In SAVE, T means: save registers 14 and 15.

GENERAL SERVICES

Dump storage and continue — SNAP

DCB=address of data control block [,TCB=address of TCB address*]

[,ID=1 to 127]

$$[\text{,SDATA=(}\left\{\begin{matrix}\text{ALL}\\\text{NUC}\\\text{TRT}\\\text{CB}\\\text{Q}\end{matrix}\right\}\left[\begin{matrix}\text{,ALL}\\\text{,NUC}\\\text{,TRT}\\\text{,CB}\\\text{,Q}\end{matrix}\right]\text{...)]}$$

$$[\text{,PDATA=(}\left\{\begin{matrix}\text{ALL}\\\text{PSW}\\\text{REGS}\\\text{SA or SAH}\\\text{JPA or LPA or ALLPA}\\\text{SPLS}\end{matrix}\right\}\left[\begin{matrix}\text{,ALL}\\\text{,PSW}\\\text{,REGS}\\\text{,SA or ,SAH}\\\text{,JPA or ,LPA or ,ALLPA}\\\text{,SPLS}\end{matrix}\right]\text{...)]}$$

SNAP		SDATA VALUES
ALL	=	all of the following fields
NUC	=	all of nucleus except trace table
TRT	=	trace table
CB	=	TCB, active RBs, JPACQ, and MSS control blocks

SNAP		PDATA VALUES
ALL	=	all of the following fields (assume SA and ALLPA)
PSW	=	Program Status Word when SNAP was issued
REGS	=	contents of general registers when SNAP was issued
SA	=	linkage information and back trace
SAH	=	linkage information only
JPA	=	all virtual storage assigned to job step
LPA	=	contents of resident reenterable load module
ALLPA	=	JPA + LPA
SPLS	=	contents of virtual storage subpools 0 - 127

$$\left[\begin{matrix}\text{,STORAGE = (starting address, ending address, . . .)}\\\text{,LIST = address of list}\end{matrix}\right]$$

Record trace data — GTRACE

DATA=address,LNG-number of bytes of data,ID=record ID

[,FID=format identifier routine]

TERMINATION

Terminate normally — RETURN

$$\text{[(register(s) 14 through 12)] [,T] }\left[\left\{\begin{matrix}\text{,RC=0 to 4095}\\\text{,RC=(15)}\end{matrix}\right\}\right]$$

In RETURN, T means: place all ones in high-order byte of save area word 4.

Terminate abnormally — ABEND

0 to 4095, [DUMP] [,STEP]

TASK CONTROL

```
Dynamically    ATTACH    ⎧ EP=entry point name                   ⎫
load and                 ⎨ EPLOC=address of entry point name+    ⎬ [,DCB=dcb address]
initiate                 ⎩ DE=address of name field of list entry⎭
execution
                         [,PARAM=(address parameter [,address parameter] ...) [,VL=1] ]

                         [,ECB=ecb address] [,ETXR=address of routine to be entered when]
                                                    subtask terminates

                         [,LPMOD=number subtracted from limit priority]

                         [,DPMOD=signed number algebraically added to dispatching priority]

                         [,TQE= ⎧YES⎫]            ⎫
                                ⎩NO ⎭             ⎪
                                                  ⎬ VS1 only
                         [,FPREGSA= ⎧YES⎫]        ⎪
                                    ⎩NO ⎭         ⎭

                         ⎡,GSPV=number         ⎤ ⎡,SHSPV=number         ⎤
                         ⎣,GSPL=address of list⎦ ⎣,SHSPL=address of list⎦

                         [,SZERO= ⎧YES⎫ ] [,TASKLIB=dcb address]
                                  ⎩NO ⎭
                         [,STAI=(exit address [,parameter list address] )]

                         [,PURGE= ⎧NONE    ⎫ ] [,ASYNCH= ⎧YES⎫]
                                  ⎨HALT    ⎬             ⎩NO ⎭
                                  ⎩QUIESCE ⎭

Delete         DETACH    address of tcb address*   [,STAE= ⎧YES⎫ ]
                                                           ⎩NO ⎭

Change         CHAP      signed number to be algebraically added to dispatching priority
priority
                         ⎡,address of tcb address⎤
                         ⎣,'S'                   ⎦   'S' indicates that the priority of the
                                                     active task is to be changed.

Change         STATUS    ⎧START⎫ [,TCB=subtask tcb address]
subtask                  ⎩STOP ⎭
dispatchibility
```

VIRTUAL STORAGE ALLOCATION

```
Allocate   GETMAIN    R,LV=lengthº [,SP=0 to 127]
storage
           GETMAIN    ⎧ ⎧EC⎫,LV=lengthº                    ⎫
                      ⎪ ⎩EU⎭                               ⎪
                      ⎪ ⎧VC⎫,LA=address of lengthº list    ⎪,A=address of specification list
                      ⎨ ⎩VU⎭                               ⎬
                      ⎪ ⎧LC⎫,LA=address                    ⎪
                      ⎩ ⎩LU⎭                               ⎭

                      [,SP=0 to 127] [,BNDRY= ⎧DBLWD⎫ ]
                                              ⎩PAGE ⎭

Release    FREEMAIN   R,LV=lengthº,A=address of storage area address* list [,SP=0 to 127]
storage
           FREEMAIN   ⎧ E,LV=lengthº ⎫,A=address of storage area address* list [,SP=0 to 127]
                      ⎨ V            ⎬
                      ⎩ L,LA=address ⎭
```

MODE OPERANDS FOR GETMAIN AND FREEMAIN

R=register type
E=single area, fixed length
V=single area, variable length
L=virtual area(s), variable length(s)
U=unconditional
C=conditional

```
Release    PGRLSE     LA=low address of area,HA=high address+1 of area
virtual
storage
```

APPENDIX C: MESSAGE ROUTING FOR MULTIPLE OPERATOR CONSOLES

ROUTING CODES

Routing codes provide the mechanism to route WTO and WTOR messages to the locations where they are needed. They indicate the functional area or areas to which a message is to be sent. If no routing code is assigned, default is to routing as if Master Console Information. These codes are not printed or displayed as part of the message text. To use routing codes effectively, the system must have the Multiple Console Support (MCS) option included at system generation. Without the MCS option, all routing codes are assigned to the one active console, except when routing code 11 is used to obtain a Write-to-Programmer message in the message output class.

Routing codes and their definitions are:

Code	Description
1	MASTER CONSOLE. This routing code is for messages that must be sent to the master console because some action is required by the master console operator, or because the message contains information considered critical to the continued operation of the system. Keep the number of messages with this attribute to a minimum.
2	MASTER CONSOLE INFORMATIONAL. This routing code is for informational messages to the master console operator. Informational messages usually require no action from the operator. If they do, that action should be at the operator's discretion.
3	TAPE POOL. See routing code 4.
4	DIRECT ACCESS POOL. The tape pool and direct access pool routing codes are for messages that contain instructions for volume handling in the tape and disk areas. Messages about error conditions which occur as a result of the operation of these devices may also be assigned one of these routing codes.
5	TAPE LIBRARY. See routing code 6.
6	DISK LIBRARY. The tape library and disk library routing codes are used for any message that specifies tape library information or disk library information.
7	UNIT RECORD POOL. This routing code is for messages about printers, punches, and card readers. Send the following classes of information to this pool: • Types of printer chains or trains required. • Carriage control tapes required. • Types of forms or cards required. • Error conditions on unit record equipment.
8	TELEPROCESSING CONTROL. Use this routing code for messages relating to teleprocessing.
9	SYSTEM SECURITY. Use this routing code for messages of interest to the system security office (such as password messages).

10	SYSTEM/ERROR MAINTENANCE. Use this routing code for any message indicating system errors or incorrectable I/O errors, and for any message associated with system maintenance.
11	PROGRAMMER INFORMATION. Use this routing code for messages of interest to the programmer. The message is included in the message class for the job and written on the system output device.
12	EMULATOR INFORMATION. This routing is for messages issued by an emulator program.
13	USER ROUTING CODE. Available for customer usage.
14	USER ROUTING CODE. Available for customer usage.
15	USER ROUTING CODE. Available for customer usage.
16	RESERVED FOR FUTURE USE.

DESCRIPTOR CODES

Descriptor codes functionally classify WTO and WTOR messages so that they may be properly presented on all consoles and deleted from display type consoles. Each WTO and WTOR message should contain one descriptor code. If no descriptor code is coded in the WTO or WTOR, no descriptor code is assumed. Descriptor codes 1 through 7 are mutually exclusive. Descriptor codes 8 and 9, however, may be used with any other descriptor code. These codes are not printed or displayed as part of the message text. To use descriptor codes (except codes 1 and 2), the system must have the Multiple Console Support (MCS) option included at system generation.

Descriptor codes and their definitions are:

Code	Description
1	SYSTEM FAILURE. This descriptor code is for messages that indicate that a catastrophic error has occurred and another IPL of the system is required.
2	IMMEDIATE ACTION REQUIRED. This descriptor code is for messages that request an immediate operator action (completion of the action is required before a task can proceed). Messages with descriptor code 2 must be deleted by a Delete Operator Message (DOM) macro instruction when the operator action has been accomplished, or the operator will have to perform the action to delete the messages.
3	EVENTUAL ACTION REQUIRED. This descriptor code is for messages requesting operator action where a task does not await completion of the action.
4	SYSTEM STATUS. This descriptor code is for messages that indicate the status of the system, such as system task status or a hardware unit status such as uncorrectable I/O errors.
5	IMMEDIATE COMMAND RESPONSE. This descriptor code is for error and nonerror messages that are written as a direct result of an operator or system command.
6	JOB STATUS. This descriptor code is for messages that indicate the status of a job or job step.

7 APPLICATION PROGRAM/PROCESSOR. This descriptor code is for messages issued by problem programs or by processors executed as problem programs. This descriptor code is the End-of-Step message deletion indicator, and all messages with this code are deleted when the job step in which they were issued is terminated.

8 OUT-OF-LINE MESSAGE. This descriptor code is used for one message or a group of one or more messages that is to be displayed out of line. If the device support cannot print a message out of line, the code will be ignored and the message will be printed in line with other messages.

9 RESPONSE TO OPERATOR REQUEST (MLWTO). This code specifies that the multiple line message is a response to an operator's request for information via the DISPLAY or MONITOR command. Specifying code 9 with a multiple line WTO will cause an MLWTO identification number to be put in the control line of the message so that the message may be canceled. If a control line does not exist, the system will provide one as follows:

IEE932I nnn

where nnn is the identification number.

10-16 RESERVED FOR FUTURE USE.

OPERANDS FOR USE BY THE SYSTEM PROGRAMMER (VS1)

For a description of the system programmer WTO operands in VS2, see the publication OS/VS2 Planning and Use Guide.

The WTO and WTOR macro instructions have two special operands, the MSGTYP and MCSFLAG operands. These operands should be used only by the system programmer who is thoroughly familiar with the Multiple Console Support (MCS) Communications Task, since improper use of these operands can impede the entire message routing scheme. These operands set flags to indicate that certain system functions must be performed, or that a certain type of information is being presented by the WTO or WTOR.

The MSGTYP and MCSFLAG operands may be specified on either the standard or list form of the WTO and WTOR macro instruction. The standard form of the WTO macro instruction is shown below.

```
┌──────────┬─────┬──────────────────────────────────────────┐
│ [symbol] │ WTO │ {'message'                     }          │
│          │     │ {('text'[,line type]),...      }          │
│          │     │  [,ROUTCDE=(number[,number],...)]         │
│          │     │  [,DESC=number]                           │
│          │     │             {N        }                   │
│          │     │  [,MSGTYP=  {Y        }]                  │
│          │     │             {JOBNAMES }                   │
│          │     │             {STATUS   }                   │
│          │     │             {ACTIVE   }                   │
│          │     │  [,MCSFLAG=(name[,name...])                │
└──────────┴─────┴──────────────────────────────────────────┘
```

'message'
    specifies that the message text is to be placed between the first and second apostrophes.

[ ] indicates optional name or operand; select one from vertical stack within { }.

ROUTCDE=
specifies the routing codes to be assigned to the message.

DESC=
specifies the descriptor codes to be assigned to the message.

MSGTYP=JOBNAMES or MSGTYP=STATUS
specifies that the message is to be routed to the console which issued the DISPLAY JOBNAMES or DISPLAY STATUS command, respectively. When the message type is identified by the operating system, the message will be routed to only those consoles that had requested the information. Omission of the MSGTYP parameter causes the message to be routed as specified in the ROUTCDE parameter.

MSGTYP=ACTIVE
specifies that the multiple-line message is in response to a MONITOR A (MN A) command and should be routed to the console that issued the command.

MSGTYP=Y or MSGTYP=N
specifies that two bytes are to be reserved in the WTO or WTOR macro expansion so that flags can be set to describe what MSGTYP functions are desired. Y specifies that two bytes of zeros are to be included in the macro expansion at displacement WTO + 4 + the total length of the message text, descriptor code, and routing code fields. N, or omission of the MSGTYP parameter, specifies that the two bytes are not needed, and that the message is to be routed as specified in the ROUTCDE parameter. If an invalid MSGTYP value is encountered, a value of N is assumed, and a diagnostic message is produced (severity code of 8).

When MSGTYP=Y, the issuer of the WTO or WTOR macro instruction that contains the MSGTYP information must set the appropriate message identifier bit in the MSGTYP field of the macro expansion (see Figure 60). Prior to executing the WTO or WTOR SVC (SVC 35), he must also set byte 0 of the MCSFLAG field in the macro expansion to a value of X'10'. This value indicates that the MSGTYP field is to be used for the message routing criteria. When the message type is identified by the system, the message is routed to all consoles that had requested that particular type of information. Routing codes, if present, are ignored.

Bit	Meaning
0	DISPLAY JOBNAMES
1	DISPLAY STATUS
2-15	Reserved for system use. Must be zeros.

Figure 60. Bit definition for MSGTYP=Y

MCSFLAG
specifies that the macro expansion should set bits in the MCSFLAG field as indicated by each name coded. Names and their corresponding bit settings are shown in Figure 61.

ROUTCDE, DESC, and MSGTYP parameter combinations are shown in Figure 62. Coding of any one of the four keyword parameters (ROUTCDE, DESC, MSGTYP, MCSFLAG) causes a new format WTO or WTOR to be generated.

Name	Bit	Meaning
----	0	
REG0	1	Message is to be queued to the console whose source ID is passed in Register 0.
RESP	2	The WTO is an immediate command response.
----	3	
REPLY	4	The WTO macro instruction is a reply to a WTOR macro instruction
BRDCST	5	Message should be broadcast to all active consoles.
HRDCPY	6	Message queued for hard copy only.
QREG0	7	Message is to be queued unconditionally to the console whose source ID is passed in Register 0.
NOTIME	8	Time is not appended to the message.
----	9	MLWTO indicator.
----	10-12	
NOCPY	13	Protect key 0 user only.
----	14-15	
Note: Invalid specifications are ignored and produce an appropriate error message.		

Figure 61. MCSFLAG parameters

	Parameter Coded				Expansion Generates			
No.	ROUTCDE	DESC	MSGTYP	MCSFLAG	ROUTCDE	DESC	MSGTYP	MCSFLAG
1	Specified	Specified	Y	Optional	Codes Specified	Codes Specified	Zeros	As Specified#
2	Specified	Specified	N	Optional	Codes Specified	Codes Specified	Field Omitted	As Specified#
3	Specified	Specified	JOBNAMES	Optional	Codes Specified	Codes Specified	X'8000'	As Specified#
4	Specified	Specified	STATUS	Optional	Codes Specified	Codes Specified	X'4000'	As Specified#
5	Specified	Specified	Omitted	Optional	Codes Specified	Codes Specified	Field Omitted	As Specified#
6	Specified	Omitted	Y	Optional	Codes Specified	Zeros	Zeros	As Specified#
7	Specified	Omitted	N	Optional	Codes Specified	Zeros	Field Omitted	As Specified#
8	Specified	Omitted	JOBNAMES	Optional	Codes Specified	Zeros	X'8000'	As Specified#
9	Specified	Omitted	STATUS	Optional	Codes Specified	Zeros	X'4000'	As Specified#
10	Specified	Omitted	Omitted	Optional	Codes Specified	Zeros	Field Omitted	As Specified#
11	Omitted	Specified	Y	Omitted*	Routing Code 2	Codes Specified	Zeros	X'8000'
12	Omitted	Specified	N	Omitted*	Routing Code 2	Codes Specified	Field Omitted	X'8000'
13	Omitted	Specified	JOBNAMES	Omitted*	Routing Code 2	Codes Specified	X'8000'	X'8000'
14	Omitted	Specified	STATUS	Omitted*	Routing Code 2	Codes Specified	X'4000'	X'8000'
15	Omitted	Specified	Omitted	Omitted*	Routing Code 2	Codes Specified	Field Omitted	X'8000'
16	Omitted	Specified	Y	REG0/QREG0	Zeros	Codes Specified	Zeros	As Specified#
17	Omitted	Specified	N	REG0/QREG0	Zeros	Codes Specified	Field Omitted	As Specified#
18	Omitted	Specified	JOBNAMES	REG0/QREG0	Zeros	Codes Specified	X'8000'	As Specified#
19	Omitted	Specified	STATUS	REG0/QREG0	Zeros	Codes Specified	X'4000'	As Specified#
20	Omitted	Specified	Omitted	REG0/QREG0	Zeros	Codes Specified	Field Omitted	As Specified#
21	Omitted	Omitted	Y	Omitted*	Routing Code 2	Zeros	Zeros	X'8000'
22	Omitted	Omitted	N	Omitted*	Routing Code 2	Zeros	Field Omitted	X'8000'
23	Omitted	Omitted	JOBNAMES	Omitted*	Routing Code 2	Zeros	X'8000'	X'8000'
24	Omitted	Omitted	STATUS	Omitted*	Routing Code 2	Zeros	X'4000'	X'8000'
25	Omitted	Omitted	Omitted	Omitted*	Field Omitted	Field Omitted	Field Omitted	Zeros
26	Omitted	Omitted	Y	REG0/QREG0	Zeros	Zeros	Zeros	As Specified#
27	Omitted	Omitted	N	REG0/QREG0	Zeros	Zeros	Field Omitted	As Specified#
28	Omitted	Omitted	JOBNAMES	REG0/QREG0	Zeros	Zeros	X'8000'	As Specified#
29	Omitted	Omitted	STATUS	REG0/QREG0	Zeros	Zeros	X'4000'	As Specified#
30	Omitted	Omitted	Omitted	REG0/QREG0	Zeros	Zeros	Field Omitted	As Specified#

* If an MCSFLAG other than REG0 or QREG0 is specified, the expansion generates the same fields except that the MCSFLAG field contains the MCSFLAG specified and the high-order bit set to 1.

# High order bit set to 1 to indicate a new format macro expansion (routing code and descriptor code fields exist).

Figure 62. ROUTCDE, DESC, and MSGTYP combinations

Indexes to systems reference manuals are :onsolidated in OS/VS Master Index, GC28-602. For additional information about any ubject listed below, refer to other publi-:ations listed for the same subject in the naster index.

OS/VS
Supervisor Services and
Macro Instructions

GC27-6979-1

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.*

How did you use this publication?

- ☐ As an introduction
- ☐ As a text (student)
- ☐ As a reference manual
- ☐ As a text (instructor)
- ☐ For another purpose (explain)____

Please comment on the general usefulness of the book; suggest additions, deletions, and clarifications; list specific errors and omissions (give page numbers):

What is your occupation?____

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

Please include your name and address 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

GC27-6979-1

Your comments, please . . .

This manual is part of a library that serves as a reference source for systems 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.

Cut or Fold Along Line

Fold Fold

First Class
Permit 40
Armonk
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 636
Neighborhood Road
Kingston, New York 12401

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)

IBM Technical Newsletter

This Newsletter No.	GN27-1400
Date	January 3, 1973
Base Publication No.	GC27-6979-1
System	OS/VS
Previous Newsletters	None

OS/VS Supervisor Services and Macro Instructions

This Technical Newsletter, a part of Release 2 of VS1, provides replacement pages for the subject publication. Pages to be inserted and/or replaced are as follows:

Cover	97,98
1,2	139-146
11-12.2	165-174
71-74	183,184
83-86	193,194
91,92	197-202

A change to the text or illustrations is indicated by a vertical line to the left of the change.

Summary of Amendments

This Technical Newsletter documents additions provided by release 2 of VS1. It also includes maintenance changes applying to both VS1 and VS2.

Please file this cover letter at the back of the manual to provide a record of changes.

IBM Corporation, Programming Publications, Dept. 636, Neighborhood Road, Kingston, N.Y. 12401

PRINTED IN U.S.A.

GC27-6979-1

Sup. Services and Macro Instr. Printed in U.S.A. GC27-6979-1

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)