File Number S360-36
GC28-6646-5

IBM Systems Reference Library

IBM System/360 Operating System Supervisor Services

The title of this manual was formerly IBM System/360 Operating System Supervisor and Data Management Services. The data management section of the book has been made a separate publication, IBM System/360 Operating System Data Management Services, GC26-3746.

This manual describes how to use the services of the supervisor. Among the services of the supervisor are program management, task creation and management, main-storage management, checkpoint and restart, and Time Sharing Option.

This book also describes the linkage conventions used by the operating system.

Intended mainly for the assembler-language programmer, this book is a guide to using the macro instructions described in IBM System/360 Operating System Supervisor and Data Management Macro Instructions, GC28-6647. This book does not discuss macro instructions used for graphics, teleprocessing, optical readers, optical reader-sorters, or magnetic character readers. These macro instructions are discussed in separate publications that are listed in the IBM System/360 Bibliography, GA22-6822.

References in this book to the forms control buffer (FCB) are applicable to the 3211 printer and are for planning purposes only.

References in this book to PCP or the primary control program are no longer applicable.

Sixth Edition (June 1971)

This publication corresponds to Release 20.1. It is a major revision of GC28-6646-4, which is now obsolete.

The changes to the book include modifications to the topics "Providing a Save Area," "Bringing the Load Module into Main Storage," "Creating the Task," "Extended-Precision Floating-Point Simulation," "Reenterable Load Modules," and "Establishing Checkpoints," and an explanation of the new operand, RET=CHNG, in the ENQ macro instruction. In addition, technical changes and clarifications have been made throughout the book, and this edition should be reviewed in its entirety.

The information in the book changes from time to time. Before using this manual with IBM systems, consult the latest IBM 360 SRL Newsletter, GN20-0360, for the editions that are current and applicable.

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

A form is provided at the back of this publication for reader's comments. 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.

This book is divided into seven sections: "Program Management," "Task Creation," "Task Management," "Program Management Services," "Main-Storage Management," "Checkpoint and Restart," and "Time Sharing Option (TSO) Services."

The "Program Management" section describes linkage conventions and ways that the supervisor can assist you in linking together the separate pieces of your program.

The "Task Creation" section describes how the system creates a task for you, and how you can create other tasks (under MVT or under MFT with subtasking). Basically, it tells how to use the ATTACH macro instruction.

The "Task Management" section deals with communication among separate tasks and with synchronization of one task with another.

The "Program Management Services" section describes several miscellaneous services that you can use in your programs. It covers the ENQ and DEQ macro instructions, timer services, communication with the operator, abnormal termination and dumps, and other miscellaneous services.

The "Main-Storage Management" section describes how to acquire and release main storage, how to share it with other tasks, and how to specify which way it is to be divided into hierarchies.

The "Checkpoint and Restart" section describes how to take checkpoints on the progress of your program, and then restart it if the system fails.

The "Time Sharing Option (TSO) Services" section describes STAX and STATUS, two macro instructions that you can use if your system has TSO.

This book assumes you have a basic knowledge of the operating system and of System/360 assembler language. Two books that contain information about these subjects are:

IBM System/360 Operating System
Introduction, GC28-6514
Assembler Language, GC28-6514

If you are using the MVT version of the control program with the time sharing option (TSO), note that this book also assumes that you understand how to use TSO. Specifically, the book assumes that you are familiar with the concepts discussed in the following books:

IBM System/360 Operating System

Time Sharing Option Command Language, GC28-6732, which describes the TSO command language that a terminal user must use to request computing services.

Time Sharing Option Guide, GC28-6698, which describes the concepts, features, and capabilities of TSO.

Time Sharing Option Guide to Writing a Terminal Monitor Program or a Command Processor, GC28-6764, which describes the programming features provided for user-written terminal monitor programs, command processors, and application programs.

If you are using the operating system without TSO, ignore the sections "Intercepting Abnormal Termination of Subtasks" and "Time Sharing Option (TSO) Services." Also ignore the TSO, PSB, and TJID operands of EXTRACT.

In the examples in this book, the macro instructions are coded in just enough detail to make the examples clear. For a complete description of all the operands and options available with any of the macro instructions discussed here, see

IBM System/360 Operating System Supervisor and Data Management Macro Instructions, GC28-6647.

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

IBM System/360

Model 91 Functional Characteristics, GA22-6907
Model 195 Functional Characteristics, GA22-6943
Principles of Operation, GA22-6821

IBM System/360 Operating System

Job Control Language Reference, GC28-6704
Linkage Editor and Loader, GC28-6538
Programmer's Guide to Debugging, GC28-6670
Service Aids, GC28-6719
Storage Estimates, GC28-6551
Supervisor and Data Management Macro Instructions, GC28-6647
System Programmer's Guide, GC28-6550

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

CONTENTS

ILLUSTRATIONS

FIGURES

The job of the supervisor is to provide the resources that your programs need in such a way that at any given time, as many resources as possible are in use. By using certain macro instructions, by specifying certain JCL parameters, and by organizing your program in certain ways, you can direct the supervisor as it goes about this job. This book tells you how to do it.

TYPES OF SERVICES AVAILABLE

The kinds of services you can request from the supervisor fall into four broad classes, with some miscellaneous services left over.

1. Program Management: Most programs are divided into segments of some sort. When these segments are separate load modules, the supervisor can be used to help the pieces communicate with each other.

   The section of this book called "Program Management" discusses save areas, addressability, and passage of control from one piece of a program to another.

2. Task Management: In some configurations of the operating system, units of work called tasks can compete with each other for resources such as CPU time.

   You can change your program's priority, you can break it into smaller units that compete with each other, and you can obtain certain information about how your tasks are progressing. You can find out how to do these things in the "Task Management" and "Task Creation" sections of this book.

3. Main-Storage Management: Frequently, a program needs more main storage all together during its run than it does at any one time. Your program might, for example, require 20,000 bytes for input buffers for one data set, and another 15,000 for buffers for another, although the two data sets need not be processed at the same time. You can use main-storage management services to get as much storage as you need, and to tell the system when you are through with the storage so someone else can use it, or so you can use it for some other purpose.

   The services available for obtaining more main storage, for freeing main storage, and for sharing main storage among several tasks are described in the "Main Storage Management" section of this book.

4. Checkpoint and Restart: If the system should fail two minutes from the end of a program that had already been running for four hours, and if the program therefore had to be run again from the beginning, a great deal of time would have been lost. The checkpoint and restart services of the supervisor allow you to take periodic checkpoints during the progress of your program, and then to restart the program at any of these checkpoints, if the system should fail.

   The "Checkpoint and Restart" section of this manual describes how to take checkpoints and then how to restart your program at a checkpoint.

5. Miscellaneous Services: The supervisor has facilities for providing dumps of main storage, communicating with the operator, handling

abnormal conditions (such as program checks), allocating serially reusable resources, and timing events. These services are discussed in the "Program Management Services" section. The supervisor also has services to use with the time sharing option (TSO). These services allow you to specify an attention exit routine and to manipulate task processing, and they are discussed in the "Time Sharing Option Services" section.

CONFIGURATIONS OF THE OPERATING SYSTEM

This book covers two major configurations of the operating system: the operating system that provides multiprogramming with a fixed number of tasks (MFT), and the operating system that provides multiprogramming with a variable number of tasks (MVT). Unless otherwise indicated in the text, the descriptions in this section apply to all configurations of the operating system; when differences arise because of operating system options, these differences are explained.

A brief description of the configurations of the operating system is given in Figure 1. This table 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 manual.

	MFT	MVT
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 Option	Standard	Standard
Identify Option	Optional	Standard
Time Option	Optional	Standard
Interval Timing Option	Optional	Standard
System Log Option	Optional	Optional

Figure 1. Summary of Characteristics and Available Options

The following discussion provides the requirements for the design of programs to be processed using the IBM System/360 Operating System. Included here are the procedures required when receiving control from the control program, the program design facilities available, and the conventions established for use in program management.

This discussion presents the conventions and procedures in terms of called and calling programs. Each program given control during the job step is initially a called program. During the execution of that program, the services of another program may be required, at which time the first program becomes a calling program. For example, the control program passes control to program A which is, at that point, a called program. During the execution of program A, control is passed to program B. Program A is now a calling program, program B a called program. Program B eventually returns control to program A, which eventually returns control to the control program. This is one of the simpler cases, of course. Program B could pass control to program C, which passes control to program D, which returns control to program C, etc. Each of these programs has the characteristics of either a called or calling program, regardless of whether it is the first, fifth or twentieth program given control during a job step.

The conventions and requirements that follow are presented in terms of one called and one calling program; these conventions and requirements apply to all called and calling programs in the system.

INITIAL REQUIREMENTS

The following paragraphs discuss the procedures and conventions to be used when a program receives control from another program. Although the discussion is presented in terms of receiving control from the control program, the procedures and conventions apply as well when control is passed directly from another processing program. If the requirements presented here are followed in each of the programs used in a job step, the called program is not affected by the method used to pass control or by the identity of the program passing control.

PROVIDING AN INITIAL BASE REGISTER

When control is passed to your program from the control program, the address of the entry point in your program is contained in register 15. This address can be used to establish an initial base register, as shown in Example 1 and Example 2. In Example 1, the entry point address is assumed to be the address of the first byte of the control section; an internal entry point is assumed in Example 2. Since register 15 already contains the entry point address in both examples, no register loading is required.

```
PROGNAME  CSECT
          USING *,15
          ...
```

Example 1. Control Section Addressability

```
            ...
PROGNAME    DS    0H
            USING *,15
            ...
```

Example 2. Internal Entry Point Addressability

SAVING REGISTERS

The first action your program should take is to save the contents of the general registers. The contents of any register your program will modify must be saved, along with the contents of registers 0, 1, 14, and 15. The latter registers may be modified, along with the condition code, when system macro instructions are used to request data management or supervisor services.

The general registers are saved in an 18-word area provided by the control program; the format of this area is shown in Figure 2. When

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 Point address)
6	Register 0
7	Register 1
8	Register 2
9	Register 3
10	Register 4
11	Register 5
12	Register 6
13	Register 7
14	Register 8
15	Register 9
16	Register 10
17	Register 11
18	Register 12

Figure 2. Save Area Format

control is passed to your program from the control program, the address of the save area is contained in register 13. As indicated in Figure 2, the contents of each of the registers must be saved at a predetermined location within the save area; for example, register 0 is always stored at word 6 of the save area, register 9 at word 15. The safest procedure is to save all of the registers; this ensures that later changes to your program will not result in the modification of the contents of a register that has not been saved.

To save the contents of the general registers, a store-multiple instruction, such as STM 14,12,12(13), can be written. This instruction places the contents of all the registers except register 13 in the proper words of the save area. (Saving the contents of register 13 is covered later.) If the contents of only registers 14, 15, and 0-6 are to be saved, the instruction would be STM 14,6,12(13).

THE SAVE MACRO INSTRUCTION

The SAVE macro instruction, provided to save you coding time, results in the instructions necessary to store a designated range of registers. An example of the use of the SAVE macro instruction is shown in Example 3. The registers to be saved are coded in the same order as they would have been designated had an STM instruction been coded. A further use of the SAVE macro instruction is shown in Example 4. The operand T specifies that the contents of registers 14 and 15 are to be saved in words 4 and 5 of the save area. The expansion of this SAVE macro instruction results in the instructions necessary to store registers 5-10, 14, and 15.

When you use the optional identifier name operand, you can code the SAVE macro instruction only at the entry point of a program. This is because the code resulting from the macro instruction with this operand requires that register 15 contain the address of the SAVE macro instruction.

PROVIDING A SAVE AREA

If any control section in your program is going to pass control to another control section and receive control back, your program is going to be a calling program and must provide another save area. Providing a save area allows the program you call to save registers without regard to whether it was called by your program, another processing program, or by the control program. If you establish beforehand what registers are available to the called program or control section, a save area is not necessary, but this is poor practice unless you are writing very simple routines.

```
PROGNAME  SAVE   (14,12)
          USING  PROGNAME,15
          ...
```

Example 3. Saving a Range of Registers

```
PROGNAME  SAVE   (5,10),T
          USING  PROGNAME,15
          ...
```

Example 4. Saving Registers 5-10, 14, and 15

Whether or not your program is going to provide a save area, the address of the save area you used must be saved. You will need this address to restore the registers before you return to the program that called your program. If you are not providing a save area, you can keep the save area address in register 13, or save it in a fullword in your program. If you are providing another save area, the following procedure should be followed:

- Store the address of the save area you used (that is, the address passed to you in register 13) in the second word of the new save area.
- Store the address of the new save area (that is, the address you will pass in register 13) in the third word of the save area you used.

The reason for saving both addresses is discussed more fully under the heading "The Dump." Briefly, save the address of the save area you used so you can find the save area when you need it to restore the registers; save the address of the new save area so a trace from save area to save area is possible.

Example 5 and Example 6 show two methods of obtaining a new save area and of saving the save area addresses. In Example 5, the registers are stored in the save area provided by the calling program (the control program). The address of this save area is then saved at the second word of the new save area, an 18 fullword area established through a DC instruction. Register 12 (any register could have been used) is loaded with the address of the previous save area. The address of the new save area is loaded into register 13, then stored at the third word of the old save area.

In Example 6, the registers are again stored in the save area provided by the calling program. The entry point address in register 15 is loaded into register 5, which is declared as a base register. The contents of register 1 are saved in another register, and a GETMAIN

```
PROGNAME  STM     14,12,12(13)
          USING   PROGNAME,15
          ST      13,SAVEAREA+4
          LR      12,13
          LA      13,SAVEAREA
          ST      13,8(12)
          ...
SAVEAREA  DC      18A(0)
```

Example 5. Nonreenterable Save Area Chaining

```
PROGNAME  SAVE    (14,12)
          LR      5,15
          USING   PROGNAME,5
          LR      3,1
          GETMAIN R,LV=72
          ST      13,4(1)
          ST      1,8(13)
          LR      13,1
          ...
```

Example 6. Reenterable Save Area Chaining

macro instruction is issued. The GETMAIN macro instruction (discussed in greater detail under the heading "Main Storage Management") requests the control program to allocate 72 bytes of main storage from an area outside your program, and to return the address of the area in register 1. The addresses of the new and old save areas are saved in the established locations, and the address of the new save area is loaded into register 13.

ESTABLISHING A PERMANENT BASE REGISTER

If your program does not use system macro instructions and does not pass control to another program, the base register established using the entry point address in register 15 is adequate. Otherwise, after you have saved your registers, establish base registers using one or more of registers 2-12. Register 15 is used by both the control program and your program for other purposes.

LINKAGE REGISTERS

Registers 0, 1, 13, 14, and 15 are known as the linkage registers, and are used in an established manner by the control program. It is good practice to use these registers in the same way in your program. As noted earlier, registers 0, 1, 14, and 15 may be modified when system macro instructions are used; registers 2-13 remain unchanged.

REGISTERS 0 AND 1: Registers 0 and 1 are used to pass parameters to the control program or to a called program. The expansion of a system macro instruction results in instructions required to load a value into register 0 or 1 or both, or to load the address of a parameter list into register 1. The control program also uses register 1 to pass parameters to your program or to the program you call. This is why the contents of register 1 were loaded into register 3 in Example 6.

REGISTER 13: Register 13 contains the address of the save area you have provided. The control program may use this save area when processing requests you have made using system macro instructions. A program you call can also use this save area when it issues a SAVE macro instruction.

REGISTER 14: Register 14 contains the return address of the program that called you, or an address within the control program to which you are to return when you have completed processing. The expansion of most system macro instructions results in an instruction to load register 14 with the address of your next sequential instruction. A BR 14 instruction at the end of any program will return control to the calling program as long as the contents of register 14 have not been altered.

REGISTER 15: Register 15, as you have seen, contains an entry point address when control is passed to a program from the control program. The entry point address should also be contained in register 15 when you pass control to another program. In addition, the expansions of some system macro instructions result in the instructions to load into register 15 the address of a parameter list to be passed to the control program. Register 15 is also used to pass a return code to a calling program.

ACQUIRING THE INFORMATION IN THE PARM FIELD OF THE EXEC STATEMENT

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

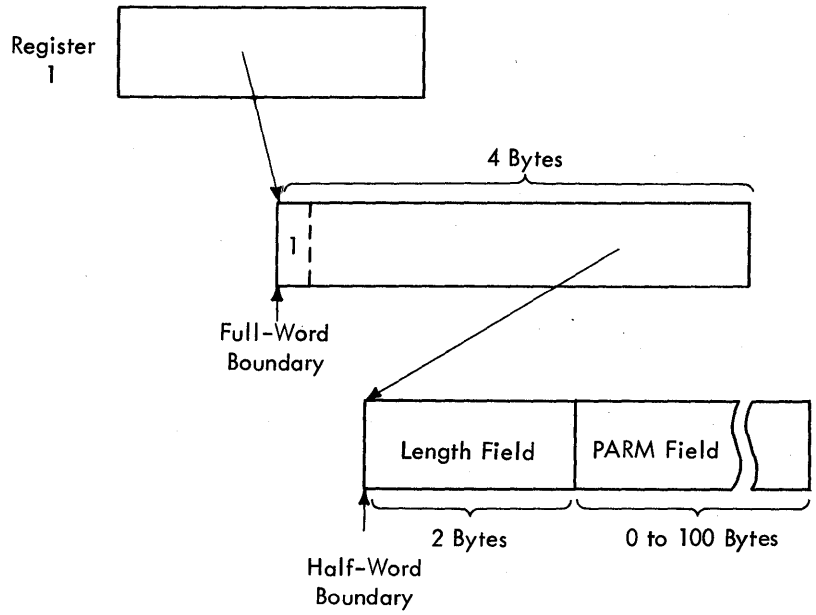


Figure 3. Acquiring PARM Field Information

address of a fullword on a fullword boundary in your area of main storage (refer to Figure 3). 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.

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 main 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 main storage all at one time. Instead, segments of the load module reuse the same area of main storage. A dynamic structure is brought into main 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.

Figure 4 summarizes the characteristics of these load module structures.

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

Figure 4. Load Module Characteristics

The following paragraphs cover the advantages and disadvantages of each type of structure, and discuss the use of each.

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 brought into the main storage all at one time by the control program. The simple structure can be the most efficient of the three structure types because the instructions it uses to pass control do not require control program intervention. However, when a program is very large or complex, the main storage area required for the load module may exceed that which can be reasonably requested. (Main storage considerations are discussed under the heading "Main Storage Management.")

PLANNED OVERLAY STRUCTURE

A planned overlay structure consists of a single load module produced by the linkage editor. The entire load module is not brought into main storage at once; different segments of the load module use the same area of main storage. The planned overlay structure, while not as efficient as a simple structure in terms of execution speed, is more efficient than a dynamic structure. When using a planned overlay structure, control program assistance is required to locate and load portions of a single load module in a library; in a dynamic structure, many load modules in different libraries may need to be located and loaded in order to execute an equivalent program.

DYNAMIC STRUCTURE

A dynamic structure requires more than one load module during execution. Each load module required can operate as either a simple structure, a planned overlay structure, or another dynamic structure. The advantages of a dynamic structure over a planned overlay 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 brought into main storage when required, and can be deleted from main storage when their use is completed.

LOAD MODULE EXECUTION

Depending on the configuration of the operating system and the macro instructions used to pass control, execution of the load modules is serial or in parallel. Execution of the load modules is always serial in an operating system with MFT without subtasking; there is only one

task in the job step. Execution is also serial in an operating system with MFT with subtasking or MVT, unless an ATTACH macro instruction is used to create a new task. The new task competes for control 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 "Task Management."

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 provide the framework around which all program interface is built. Knowledge of the information contained in the section "Addressing -- Program Sectioning and Linking" in the Assembler Language publication is required.

PASSING CONTROL WITHOUT RETURN

A control section is usually written to perform a specific logical function within the load module. Therefore, there will be occasions when control is to be passed to another control section in the same load module, and no return of control is required. 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.

INITIAL REQUIREMENTS

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 will not make the return to the calling program, the return address must be passed to the control section that will make the return. In addition, the contents of registers 2-12 must be unchanged when your program eventually returns control, so these registers must also be restored.

If control were being passed to the next entry point from the control program, register 15 would contain the entry point 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 full words 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 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 main storage area required 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 name.

An example of proper register loading and control transfer is shown in Example 7. 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 the use of a parameter list is shown in Example 8. Early in the routine the contents of register 1 (that is, the address of the fullword containing the PARM field address) were stored at the fullword PARMADDR. Register 13 is loaded with the address of the old save area, which had been saved in word 2 of the new save area. The contents of register 14 are restored, and register 15 is loaded with the entry point 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

```
            ...
            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)
```

Example 7. 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)
```

Example 8. Passing Control With a Parameter List

sections, and expect to receive control back. An example of this type of control section is a "monitor" portion of a program; 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.

INITIAL REQUIREMENTS

Registers 15 and 1 are used in exactly the same manner as they were used when control was passed without return. Register 15 contains the entry point address in the new control section and register 1 is used to pass a parameter list.

Using the standard convention, register 14 must contain the address of the location to which control is to be passed when the called control section completes processing. This time, of course, it is a location in the current control section. 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 available for passing control to another control section and providing for return of control. One is merely 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 point and return point 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. Example 9 and Example 10 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.

Use of an inline parameter list and an answer area is also illustrated in Example 9. 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 Example 9 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 full 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 Example 10 provides the same functions as the instructions in Example 9. 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.

```
            ...
            L     15,NEXTADDR     Entry point 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
```

Example 9. Passing Control With Return

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

Example 10. Passing Control With CALL

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 Example 10 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 the instructions necessary to 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 parameters as you need as address parameters to be passed, 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 and STOW macro instructions in the Supervisor and Data Management Macro Instructions book.

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 code shown in Example 11 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 Example 12, could be used to pass control to the proper routine.

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

Example 11. 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
          ...
```

Example 12. Return Code Test Using Branching Table

HOW CONTROL IS RETURNED

In the discussion of the return under the heading "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; return of control should always be passed 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.

Example 13 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 proceeds based on the number of out-of-tolerance conditions encountered. The coding shown in Example 13 causes register 13 to be loaded with the address of the save area this control section used, then reloads register 14 with the proper 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 the instructions necessary to restore a designated range of registers, provide the proper return code value 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

```
          ...
          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'
```

Example 13. Establishing a Return Code

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. This is usually not done because it requires too much main storage.

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 value before you code the RETURN macro instruction, you can specify the return code value in the RETURN macro instruction, or you can reload register 15 from the save area.

The code shown in Example 14 provides the same result as the code shown in Example 13. Registers 13 and 14 are reloaded, and the proper value is established in register 15. The RETURN macro instruction causes registers 2-12 to be reloaded, and control to be passed 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 value, and the contents of register 15 are not to be altered.

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

```
          ...
          L       13,4(13)          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'
```

Example 14. Use of the RETURN Macro Instruction

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

Example 15. RETURN Macro Instruction With Flag

RETURN TO THE CONTROL PROGRAM

The discussion in the preceding paragraphs has covered passing control within one load module, and has been based on the assumption that the load module was brought into main 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 is returned to the 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 the following job steps, if any, should be executed.

PASSING CONTROL IN A PLANNED OVERLAY STRUCTURE

A complete discussion of the requirements for passing control in an overlay environment is provided in the Linkage Editor and Loader manual.

PASSING CONTROL IN A DYNAMIC STRUCTURE

The discussion of passing control in a simple structure has provided the necessary background for the discussion of passing control in a dynamic structure. Within each load module, control should be passed as in a simple structure or planned overlay structure. If you can determine which control sections will make up a load module before you code the control sections and if they will fit in the main storage available, 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 intervention and possibly increased execution time.

BRINGING THE LOAD MODULE INTO MAIN STORAGE

The load module containing the entry point name you specified on the EXEC statement is automatically brought into main storage by the control program. Any other load modules you require during your job step are brought into main storage by the control program as a result of specific requests for dynamic acquisition; these requests are made through the use of the LOAD, LINK, ATTACH, or XCTL macro instructions. The following paragraphs discuss the proper use of these macro instructions.

LOAD MODULE LOCATION

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 necessary 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 necessary data control block and issues the OPEN macro instruction to logically connect the library to your program.

- In systems with MVT, 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 on 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 will be the first library searched when Task C requests that a module be brought into main storage.

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

- A private library is established 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 necessary 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 Section II: Data Management Services.

Operating systems with MFT or MVT may already have some of the load modules from the link library in main storage in an area called the resident reenterable module area (optional in MFT) or the link pack area (MVT). The contents of these areas are determined at Initial Program Loading time, and will vary depending on the requirements of your installation. In an operating system with MVT, the link pack area contains frequently used, reenterable load modules from the link library along with data management load modules; these load modules can be used by any job step in any job. When it is started, TSO extends the link pack area. In an operating system with MFT, 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 load modules you request are brought into your area of main storage, and are available to any task in your job step. For systems with MVT and MFT with subtasking, the portion of your area containing the copies of 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 (MVT and MFT with subtasking), the libraries, and the link pack area (MVT) or the resident reenterable module area (MFT). If a copy of the load module is found in one of the pack areas, the control program determines whether or not that copy can be used, based on criteria discussed under the heading "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.

The order in which the libraries and pack areas are searched depends on whether the system is MVT or MFT, and upon the operands used in the macro instruction requesting the load module. The operands that define the order of the search are the EP, EPLOC, DE, and DCB operands. 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 blocks for the link library, the job or step library, or the task library.

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

The EP and EPLOC operands require the least effort on your part; you provide only the entry point name, and the control program searches for a load module having that entry point name. Figure 5 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, task, job, or step library. In a system with MVT, 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 ascendents. The job or step library is then searched, followed by the link library. In a system with MFT, the job or step library is the first searched, followed by the link library. 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 point 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

MFT	MVT
The partition is searched.	The job pack area of the region is searched for an available copy.
	The requesting task's task library and all the unique task libraries of its direct ascendants are searched.
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 step library or the job library (if any) is searched. If both libraries are specified, the job library is not searched.	The link pack area is searched.
The link library is searched.	The link library is searched.

Figure 5. Search for Module, EP or EPLOC Operands With DCB=0 or DCB Operand Omitted

the unwanted version is in PDS2, a step library consisting of these data sets should be defined as follows:

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

If, however, the first version in the job or step library has been previously loaded and the version in the link library or the second version in the job library is desired, the DCB operand must be coded on the macro instruction.

This is not the case for task libraries. Extreme caution should be used when specifying module names in unique task libraries, because duplicate names may lead to the wrong module being given to the task requesting that the module be brought into main storage. Once a module has been loaded, the module name is known to all tasks in the region and a copy of that module will be 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 operands, this time in conjunction with the DCB operand. You would 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 6.

Searching a job step, or task 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 compile) will not require a step library, and therefore will not require any unnecessary search in 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.

MFT	MVT
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 6. Search for Module, EP or EPLOC Operands With DCB Operand Specifying Private Library

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.

To save time, the BLDL macro instruction used must obtain directory entries for more than one entry point 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 point name requested in a designated location in main 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 exact 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 7 for the link, job, step, and private libraries.

The preceding discussion of the search is based on the premise that the entry point name you specified is the member name. When you are using an operating system with MFT, the same search results from specifying an alias rather than a member name. When you are using an operating system that includes MVT, the control program checks if the entry point name is an alias 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, 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 main 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 main 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 main storage, followed by loading the load module into main storage. If you know the location of the load module, you should use the operands in your macro

instruction that eliminate as many of these unnecessary searches as possible, as indicated in Figure 5, Figure 6, and Figure 7. Examples of the use of these figures are shown in the disucssion of passing control.

USING AN EXISTING COPY

The control program will use a copy of the load module already in the link pack area or 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 or not 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 point 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 will protect you from obtaining an unusable copy of a load module as long as you always "formally" request a copy using these macro instructions (or the EXEC statement); if you ever 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 MVT: If you are using an operating system with MVT, all reenterable modules (modules designated as reenterable using

MFT	MVT
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 7. Search for Module Using DE Operand

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 will always be used for a LOAD macro instruction. If the copy is in use, however, and the request is made using a LINK, ATTACH, or XCTL macro instruction, the task requiring the load module is placed in a wait condition until the copy is available. A LINK macro instruction should not be issued for a serially reusable load module currently in use for the same task; the task will be abnormally terminated. (This could occur if an exit routine issued a LINK macro instruction for a load module in use by the main program.)

If the load module is nonreusable, 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.

MFT Systems With Subtasking: If you are using an MFT system with subtasking, the LOAD macro instruction enables all tasks in a partition to share the same copy of a reenterable module invoked by a previous LOAD macro instruction. If the reenterable module is again invoked by a LINK, XCTL, or ATTACH macro instruction and a previous request is still active, a new copy of the module will be brought into main storage.

MFT Systems Without Subtasking: If you are using an operating system with MFT, the macro instruction used to request the load module also determines if an existing copy can be used. If a LOAD macro instruction is issued, an existing copy is always used to satisfy the request, without regard to the reusability designation or the current status of the copy. However, if an ATTACH, LINK, or XCTL macro instruction is issued, an existing copy is used only if that copy was brought into main storage as a result of a request using a LOAD macro instruction and the copy is not in use; otherwise, a new copy is brought into the job pack area.

MFT Systems with the Resident Reenterable Module Area Option: If you are using an operating system with the MFT resident reenterable module area option, and you request use of a module by issuing an ATTACH, LINK, LOAD, or XCTL macro instruction, the supervisor will search the resident reenterable module area for a copy of the module before fetching a new copy into main storage.

USING THE LOAD MACRO INSTRUCTION

The LOAD macro instruction is used to ensure that a copy of the specified load module is in main storage in your job pack area if it is not preloaded into the link pack area. When a LOAD macro instruction is issued, the control program searches for the load module as discussed previously, and brings a copy of the load module into the job pack area if required. When the control program returns control, register 0 contains the main storage address of the entry point specified for the requested load module. 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 main 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 main storage area containing the copy is made available; the copy is never reused after the responsibility count established by LOAD macro instructions reaches zero.

Copies of load modules are not added to or deleted from the link pack area; LOAD and DELETE macro instructions issued for load modules already in the link pack area result in returns indicating successful completion, however.

PASSING CONTROL WITH RETURN

The LINK macro instruction is used to pass control between load modules and to provide for return of control. In an operating system with MFT without subtasking, the ATTACH macro instruction is executed in a similar manner to the LINK macro instruction. 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 point address and register 14 contains the return point 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 point name and possibly some library information using the EP, EPLOC, or DE, and DCB operands. But you have to get this entry point 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 point, 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.

Examples 16 and 17 show the coding of a LINK macro instruction used to pass control to an entry point in a load module. In Example 16, the load module is from the link, job, or step library; in Example 17, the module is from a private library. Except for the method used to pass control, this example is similar to Examples 9 and 10. 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 5.

Examples 18 and 19 show the use of the BLDL and LINK macro instructions to pass control. Assuming 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 main 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.)

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 main 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 Example 19 can now be written. Note that the DE operand refers to the name field, not the list description field, of the directory entry.

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

Example 16. 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)
```

Example 17. 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
```

Example 18. Use of the BLDL Macro Instruction

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

Example 19. The LINK Macro Instruction With a DE Operand

USING THE ATTACH MACRO INSTRUCTION (MFT WITHOUT SUBTASKING)

This discussion applies only if you are using an operating system with MFT without subtasking. In an operating system with MVT or with MFT with subtasking, you use the ATTACH macro instruction to cause parallel execution, as discussed under the heading "Task Creation."

The ATTACH macro instruction performs exactly the same functions as the LINK macro instruction, and should be used in exactly the same way. You should use the ATTACH macro instruction only when coding for upward compatibility with an operating system that includes MVT. There are two additional optional operands provided with the ATTACH macro instruction: the ECB and ETXR operands. They provide a means of communicating between tasks from the same job step when they are used in an operating system with MVT. They do not provide this service in the other configurations of the operating system because there is only one task for each job step. If your program is ever to be run in a system with MVT, the use of these operands in the other configurations provides an opportunity to check the routines associated with these operands. Refer to "Task Management" for a discussion of the ECB and ETXR operands if this is the case. You may find other uses for these operands in your current system.

The ECB operand allows you to specify the address of an event control block, a fullword which will be used by the control program to inform you of the completion of the called load module. The return code from the called load module will also be placed in the fullword. For a complete discussion of the event control block and its purpose, see "Task Management."

The ETXR operand provides the means of specifying an end-of-task exit routine to be given control following the completion of the called load module. This exit routine must be in main storage when it is required. The routine is given control by the control program and must return control to the control program using the address in register 14. The control program then returns control to the instruction following the ATTACH macro instruction.

USING CALL OR BRANCH AND LINK

You can save time by passing control to a load module without using the control program. Passing control without using the control program is performed as follows: issue a LOAD macro instruction to obtain a copy of the load module, preceded by a BLDL macro instruction if you can shorten the search time by using it. The control program returns the address of the entry point in register 0. 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 point 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 you.

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 current 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 current 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 "Program Management Services" 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 when using an operating system with MFT or MVT.

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. If you are using an operating system with MVT or with MFT with subtasking, you can ensure that you always get a new copy by using a LINK macro instruction or by doing as follows:

- Issue a LOAD macro instruction before you pass control.
- Pass control using a branch or a branch and link instruction or a CALL macro instruction only.
- Issue a DELETE macro instruction as soon as you are through with the copy.

If you are using an operating system with MFT without subtasking, you should perform the same three steps indicated above, and also make sure that you do not require a second use of the load module before completion of the first use.

HOW CONTROL IS RETURNED

The return of control between load modules is exactly 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 establishing 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 the contents of registers 2-13 to be unchanged, the contents of register 14 to be the return point address, and optionally, the contents of register 15 to be a return code. The return of control can be made using a branch instruction or the RETURN macro instruction. If control was passed without using the control program, that is all there is to it. However, if control was originally passed using the control program, the return of control is to the control program, then to the calling program. The action taken by the control program is discussed in the following paragraphs.

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 main storage as long as it was required. The return of control indicates to the control program that this use of the copy is completed, so the responsibility count is decremented by one. If you are using an operating system with MFT, the main storage area containing the copy is made available when the responsibility count reaches zero. If you are using an operating system with MVT, the copy is retained when the responsibility count reaches zero if all three of the following requirements are met:

- The load module attributes are serially reusable or reenterable.
- The count was not reduced to zero because of a DELETE macro instruction.
- The main storage area is not required for other purposes.

If control was originally passed using an ATTACH macro instruction (MFT without subtasking), the control program takes the following action:

- If the ECB operand was specified, the control program posts the return code in the indicated fullword.
- If the ETXR operand was specified, the control program passes control to the designated address, using register 15 to contain the entry point address, and register 14 to contain the return point address (to the control program). When the exit routine returns control, the control program passes control to the instruction following the ATTACH macro instruction without modifying the

contents of any register except register 14. Register 15 does not, in this case, contain the return code.

If the ETXR operand was not specified, or if the LINK macro instruction was used to pass control, the control program only places the return point address into register 14, and passes control to that address. No other register contents are modified.

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 non-reusable 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 point 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. Registers 1 is used to pass parameters in a parameter list. A branch instruction is issued to pass control to the address in register 15.

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 also 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 point 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 needs your base register to be addressable, 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 contained in register 1. In this case, however, the parameter list must be established in a portion of main 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, establishes the entry point address in register 15, saves the address passed in register 14 and replaces it with a new return point address within the control program, 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.

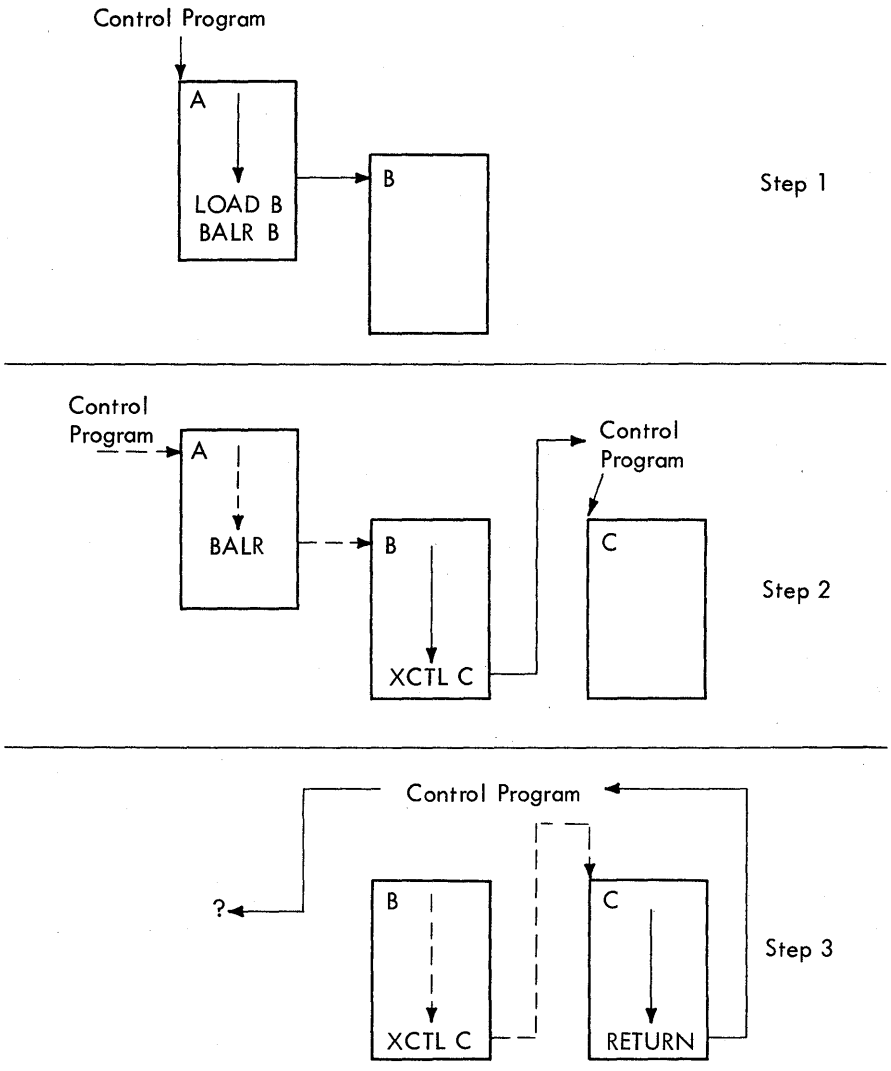


Figure 8. Misusing Control Program Facilities

Figure 8 shows in detail 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, independent 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 8 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 main storage.

TASK CREATION

In any configuration of the operating system, one task is created by the control program as a result of initiating execution of the job step. In an operating system with MFT without subtasking, only the control program can create tasks; your program cannot create tasks.

In an operating system with MVT or with MFT with subtasking, 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 under MVT or under MFT with subtasking. The benefits of a multiprogramming environment are still available even with only one task in the job step; work is still being performed when your task is unable to use the system while 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 one of the following return codes in register 15:

Hexadecimal Code	Meaning
00	Indicates successful completion of the ATTACH request.
04	Indicates that the ATTACH macro instruction was issued in a STAE exit routine.
08	Indicates that sufficient main storage was not available to schedule the exit routine as specified by the STAI operand. The subtask has not been successfully created.
0C	Indicates that the exit routine or parameter list address specified in the STAI operand was invalid. The subtask has not been successfully created.
10	Indicates that storage for the STAI request is not available for the propagation of STAIs from the mother to the daughter task.

The entry point in the load module to be given control when the subtask becomes active is specified in the same way as 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 the section titled "Program Management." Parameters can be passed to the subtask using the PARAM and VL operands, also described in "Program Management." Ownership of subpools is transferred or shared using the GSPV, GSPL, SHSPV, and SHSPL operands discussed in "Main Storage Management." The only additional operands are those dealing with the priority of the subtask, the operands that provide for communication between tasks, and the TASKLIB operand.

The TASKLIB operand is used to specify the address of an opened data control block (DCB) for a job library to be searched for the entry point name of the module being attached and for the subsequent modules accessed by the subtask. If the TASKLIB operand is not specified, the job library DCB address from the attaching task's TCB is propagated to the subtask.

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

TASK PRIORITY

In a system with MVT or MFT with subtasking, tasks compete for control on the basis of priority. When a task is created, it is assigned a priority that can later 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 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:

$$DPRTY=(value_1,value_2)$$

where $value_1$ and $value_2$ are both integers from 0 to 15. Dispatching priority is then computed as follows:

$$\text{Dispatching Priority} = (value_1 \times 16) + value_2$$

For example, if $value_1$ is 6 and $value_2$ is 4:

$$\text{Dispatching Priority} = (6 \times 16) + 4 = 100$$

Note that you can specify any dispatching priority from 0 (DPRTY=(0,0)) to 255 (DPRTY=(15,15)).

If you omit the DPRTY parameter for a job step, the initial dispatching priority of the job step task is determined by the job priority. You 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.

When you specify job priority, you 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

This is the same priority that would result from coding the parameter DPRTY=(value,11).

To specify a dispatching priority equal to that which would be computed from a given job priority, you can specify:

DPRTY=$value_1$

where $value_1$ is the job priority. The omitted $value_2$ has an assumed value of 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 later in "Obtaining Information from the Task Control Block."

To summarize, the initial dispatching priority of the job step task can be determined four ways:

1. It can be specified directly through the DPRTY parameter of the EXEC statement.

2. It can be specified indirectly through the PRTY parameter of the JOB statement.

3. It can be determined by default when the PRTY and DPRTY parameters are both omitted.

4. It can be determined by the operations staff, overriding your own specifications.

Whichever way it is determined, the initial dispatching priority is always the limit priority for the job step task.

The job step task can lower its initial 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. Of course, 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 using 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 new task. 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 new task, unless the number is greater than the limit priority. In that case, the limit priority value is used as the dispatching priority.

There are no absolute rules for assigning priorities to tasks and subtasks. Priorities should be assigned on the basis that tasks of higher priority will be given control when competing with tasks of lower 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. When the input/output operation has completed, the higher priority tasks will get control so that the next operation can be started. In addition, if one or more subtasks must be completed before the originating task can proceed beyond a certain point, the subtasks that must be completed should be assigned a priority which will eliminate as much as possible a long wait time in the originating task.

Since tasks from other job steps are competing for control, the priority initially established for the subtasks may be too high or too low to properly process the job step. To correct this, the priorities of these tasks can be changed after the tasks have been created by using the CHAP macro instruction. The EXTRACT macro instruction, discussed later, can be used to determine the current dispatching and limit priorities 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.

TIME SLICING

Time slicing is an optional feature of the operating system with MFT or MVT. It enables tasks that are members of the "time-slice group" to share control of the CPU. When a member of the time-slice 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.

MFT SYSTEMS WITHOUT SUBTASKING

At system generation, your installation designates certain contiguous main storage partitions for time slicing. Your tasks (job steps) are members of the time-slice group if your job is assigned to one of these partitions. You control partition assignment through the CLASS parameter of your JOB statement.

MFT SYSTEMS WITH SUBTASKING

Any task or subtask is considered 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.

During execution, a task or subtask can use the CHAP macro instruction to designate itself as 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, a parent task can use the ATTACH or CHAP macro instructions to designate a subtask as a member of the time-slicing group if the limit priority of the parent task is equal to or greater than the lowest dispatching priority of the time-slicing group.

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. The highest and lowest dispatching priorities of a partition are given in Figure 9. The dispatching priorities indicated in the figure must be decremented by 1 for each of the following functions that are included in the system:

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

Figure 9. Determining Partition Dispatching Priorities

- System Log
- System Management Facility
- I/O Recovery Management Support

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.

MVT 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-slice 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 slice 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 slice 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-slice 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-slice group. However, it can create lower priority subtasks that are members of the time-slice 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-slice group when time slicing is not actually intended.

The task management information in this section is required only for establishing communications among tasks in the same job step, and therefore applies only to operating systems with MVT or with MFT with subtasking. The relationship of tasks in a job step is shown in Figure 10.

The horizontal lines in Figure 10 divide the tasks into various levels. These levels have no relation to task priorities; they serve only to separate originating tasks and subtasks. Tasks A, B, A1, A2, A2a, B1, and B1a are all subtasks of the job step task; Tasks A1, A2, and A2a are subtasks of Task A. Tasks A2a and B1a are the lowest level tasks in the job step. Although Task B1 is at the same level as Tasks A1 and A2, it is not considered a subtask of Task A.

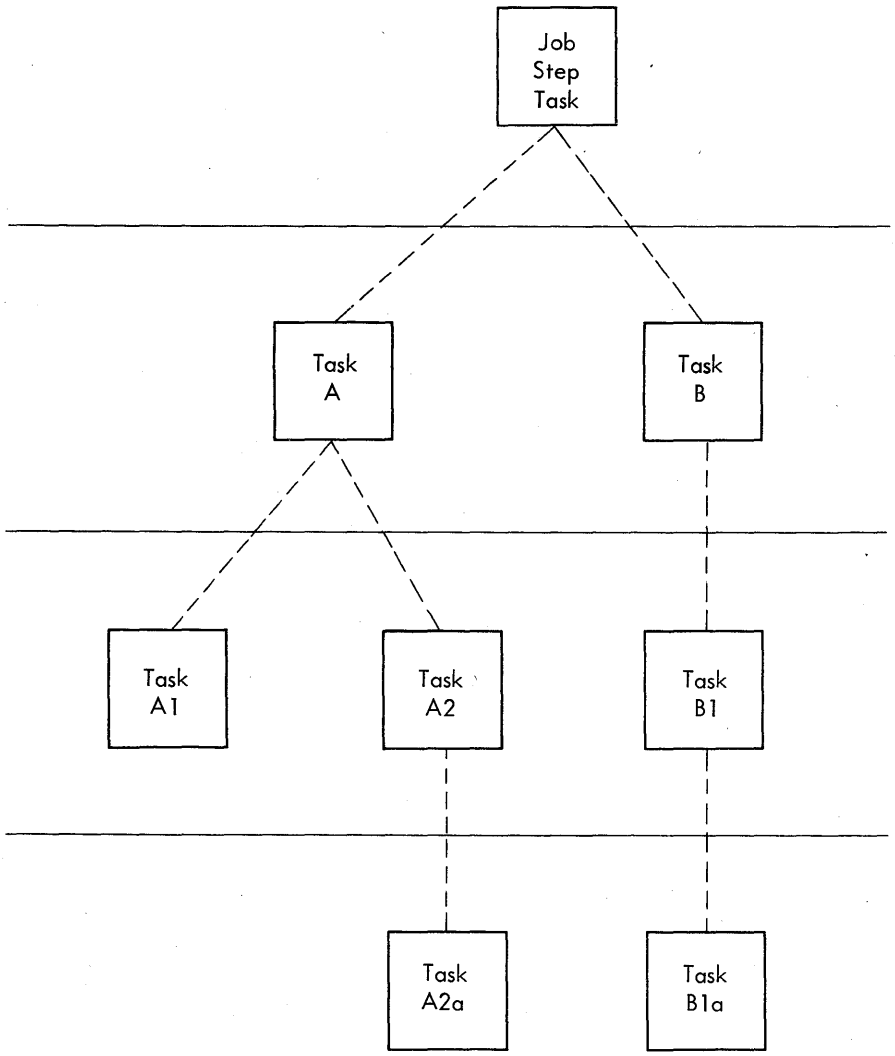


Figure 10. Task Hierarchy

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

All of the tasks in the job step compete independently for control; 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, you will usually require some communication and constraints between tasks, 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.

TASK AND SUBTASK COMMUNICATIONS

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 either the ECB or ETXR operands, or both, are coded in the ATTACH macro instruction, the task control block of the subtask is not removed from the system when the subtask is terminated. The originating task must remove the task control block from the system after termination of the subtask. This is accomplished by issuing a DETACH macro instruction. The task control blocks for all subtasks must be removed before the originating task can terminate normally.

The ETXR operand specifies the address of an end-of-task exit routine in the originating task to be given control when subtask being created is terminated. The end-of-task routine is given control asynchronously after the subtask has terminated, and must be in main storage when it is required. After the control program terminates the subtask, the end-of-task routine specified when the subtask was created is scheduled to be executed. The routine competes for control on the basis of 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.

The end-of-task routine can issue an EXTRACT macro instruction specifying the task control block of the terminated subtask. The address of that task control block is contained in register 1 when the routine is given control. The EXTRACT macro instruction, discussed under the heading "Obtaining Information From the Task Control Block," can be used to obtain such information as floating-point register contents and completion code. 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, the event control block contains the completion code specified for the subtask.

If neither the ECB nor ETXR operands are specified in the ATTACH macro instruction, the task control block for the subtask is removed from the system when the subtask is terminated. No DETACH macro instruction is required. Use of the task control block in a CHAP, EXTRACT, or 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.

TASK SYNCHRONIZATION

Task synchronization requires some planning on your part 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.

The control block used with both the WAIT and POST macro instructions is the event control block. An event control block is a fullword on a fullword boundary and is shown in Figure 11.

An event control block is 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 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 5. 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.

When an event control block is originally created, bits 0 and 1 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 issued. However, if the bits are set to zero before posting the ECB, 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.

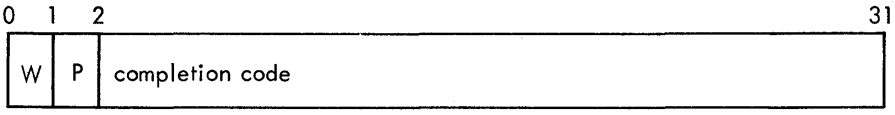


Figure 11. Event Control Block

The control program provides a set of optional services which are available to your program through the use of macro instructions. The following paragraphs discuss each of these services and the way to obtain them. The proper use of any of these services results in an improved and more efficient program; the misuse or overuse of the services wastes main storage and execution time.

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 main storage. Once a copy has been brought into main storage, however, additional entry points can be provided for the load module, subject to the following restrictions:

- The "identify" option must have been included in the operating system during system generation (standard in an operating system with MVT, optional with the other configurations of the operating system).
- 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, or
  - 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. A further restriction exists for an operating system with MFT: 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 main 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 would cause errors. The control program checks the names of all load modules currently in the link pack area and the job pack area 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 unintentionally.

The added entry point can be used only in an ATTACH macro instruction when you are using an operating system with MFT, and can be used in an

ATTACH, LINK, LOAD, DELETE, or XCTL macro instruction in an operating system with MVT. The added entry point can be used in the performance of any task in the job step; if the copy is in the link pack area, the entry point can be used in the performance of any task in the system.

The added entry point is available for as long as the copy is retained in main 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 code.

ENTRY POINT AND CALLING SEQUENCE IDENTIFIERS

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 Programmer's Guide to Debugging.

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 point address. The high-order two bytes of the fullword form a NOP instruction.

USING A SERIALLY REUSABLE RESOURCE

The example of a serially reusable resource already encountered was a load module that was designated serially reusable. In the discussion of the serially reusable load module it was emphasized that simultaneous uses of the load module must be prevented. This is true for any serially reusable resource when one or more of the users will modify the resource.

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

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 required by more than one task, the ENQ macro instruction is provided to ensure use of the resource in a serial manner. 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 in an operating system with MFT or MVT.

The ENQ macro instruction requests the control program to assign control of a resource to the active task. The control program determines the current 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 will add the job step identifier to the rname so that no duplicate qname and rname combination will be 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 and perhaps published. 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 resource for a task by coding either "E" or "S", respectively, 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, as discussed previously, 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 essentially constructs a list for each qname and rname combination it receives in an ENQ macro instruction, and makes an entry in the list representing the task which is active when the ENQ macro instruction is issued. The entry is made 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 entry representing the task is placed on the list in the order the request is 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 based on two factors:

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

The control program uses these two factors in determining whether control of a resource can be allocated to a task, as indicated below. Figure 12 shows the current 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, respectively. The task represented by the first entry on the list is always given control of the resource, so the task represented by ENTRY 1 (Figure 12, Step 1) is assigned the resource. The request which established ENTRY 2 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.

Eventually control of the resource is released for the task represented by ENTRY 1 and the entry is removed from the list. As shown in Figure 12, Step 2, ENTRY 2 is now first on the list, and the corresponding task is assigned control of the resource. Because the request which established ENTRY 2 was for exclusive control, the tasks represented by all the other entries in the list are kept in the wait condition.

Figure 12, Step 3 shows the status of the list after control of the resource is released for the task represented by ENTRY 2. Because ENTRY 3 is now at the top of the list, the task represented by ENTRY 3 is

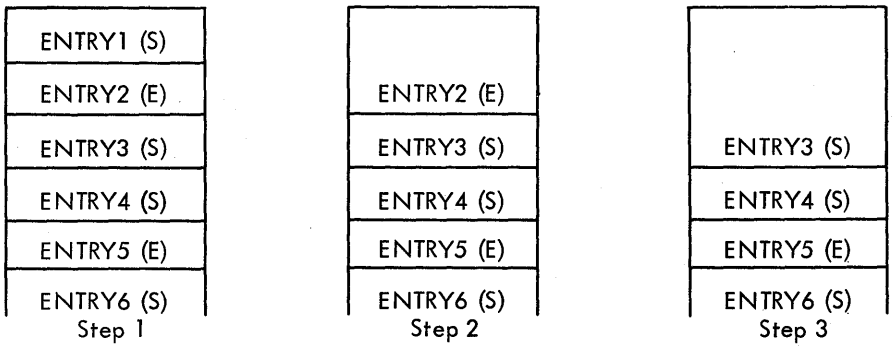


Figure 12. ENQ Macro Instruction Processing

given control of the resource. ENTRY 3 indicated the resource could be shared, and, because ENTRY 4 also indicated the resource could be shared, ENTRY 4 is also given control of the resource. In this case, the task represented by ENTRY 5 will not be given control of the resource until control has been released for both the tasks represented by ENTRY 3 and ENTRY 4. The remainder of the list is processed in the same manner.

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 the corresponding entry is the first entry in the list.
- If the request is for shared control, the task is given control either when the corresponding entry 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 multiple 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.

PROPER USE OF 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 proper use are given in the following paragraphs.

DUPLICATE REQUESTS

A duplicate request occurs when an ENQ macro instruction is issued to request a resource if a task has already been assigned control of that resource or if a task is already waiting for 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 plan the logic of 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 CONTROL OF THE RESOURCE

The DEQ macro instruction is used to release control of 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. Control of 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 control of the resource, so that other tasks can be performed. If you return to the control program at the end of processing for any task which is still assigned control of a resource, the resource is released automatically; however, in a system with MVT, the task is abnormally terminated.

CONDITIONAL AND UNCONDITIONAL REQUESTS

The normal use of the ENQ and DEQ macro instruction is to make unconditional requests. These are the only requests we have 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 more careful program design or through the use of the RET operand in the ENQ or 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 control or release of control.

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 indicates an entry for the same task already exists in the list. 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 when used to determine if the task has already been assigned control of the resource. It is less useful when used to determine the current 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 the return codes are checked by your program and another ENQ macro instruction issued, 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 an entry has been made on the list and the task has been 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; no entry is made in the list. A return code of 8 indicates an entry for the same task already exists in the list. RET=USE can be best used when there is other processing that could be performed without using the resource. You would not want to wait for the resource as long as there was other work that you could do.

RET=CHNG indicates to the control program that the caller wishes to have exclusive control of the resource for which he is already enqueued. A return code of 0 indicates that the resource is immediately available and has been assigned to the exclusive control of the caller. Either the caller was already enqueued with the exclusive attribute, or the requested change from shared to exclusive 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 enqueued for the resource when he requested the attribute change. Although this is an error condition, control is returned to the user.

RET=HAVE is used in both the ENQ and DEQ macro instructions. An ENQ macro instruction is processed as a normal request for control unless an entry for the same task already exists. A return code of 8 indicates an entry for the same task already exists in the list. A return code of 0 indicates that the task has been assigned control of the resource. A DEQ macro instruction is processed as a normal request to return control unless the task does not have control of the resource. A return code of 0 indicates that control of the resource has been released. A return code of 8 indicates that the task does not have control of 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 other to complete, yet neither task can gain access to the resource it needs to complete processing. An example of an interlock situation is shown in Figure 13. 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 developed if Task B issued a single request for multiple resources M and N prior to Task A's second request. However, the interlock would not have developed 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 have not contributed to the conditions which caused the interlock.

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

- 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 control for one before requesting control for the next.

- Request shared control as much as possible. If the entries in the lists shown in Figure 13 had indicated shared requests, there would have been no interlock. This does not mean you should indicate a request for shared control when you will modify the resource. It does mean that you should analyze your requirements for the resources carefully, and not make requests for exclusive control when requests for 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 access to the resource. For example, instead of coding the two ENQ macro instructions shown in Example 20, the one ENQ macro instruction

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 13. Interlock Condition

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

Example 20. Two Requests for Two Resources

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

Example 21. One Request for Two Resources

shown in Example 21 could be coded. If all requests were made in this manner, it would avoid the interlock shown in Figure 13. All of the requests for one task would be processed before any of the requests for 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 resource, then the pair of resources can be defined as one resource in the ENQ and DEQ macro instructions. This procedure can be used for any number of resources that are always used in conjunction. 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 for control should always be made in the order of A, B and C. In this manner 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 condition. 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 situation was developing, of course, this would be an unnecessary waste of execution time. The solution to the interlock problem in all cases requires the cooperation of all the users of the resources.

OBTAINING INFORMATION FROM THE TASK CONTROL BLOCK

Most of the information available from the task control block is useful primarily in task management. The following paragraphs discuss the information available and how to obtain it. How you use the information provided depends on the application of your program.

The EXTRACT macro instruction is used to obtain information from the task control block (TCB), the command scheduler control block (CSCB), and the interruption request block (IRB). The full power of the EXTRACT macro instruction is available (and needed) only in an operating system with MVT or MFT with subtasking. However, a limited amount of information can be obtained through the use of the EXTRACT macro instruction with the other configurations of the operating system.

Information can be obtained from the TCB, CSCB, and IRB for the active task or any of its subtasks. The following information can be requested:

TCB

- The address of the general and floating point register save areas. These are the save areas used by the control program when the task is not active.
- The limit and dispatching priorities of the specified task.
- The completion code if the task has been terminated. If the specified task has not been terminated, the completion code value is set to zero.
- The address of the time sharing flags field (TCBTSFLG) and the protected storage control block (PSCB) from the job step control block (JSCB). This information can be obtained only when using EXTRACT in an operating system with the time sharing option (TSO).

CSCB

- The addresses of the task input/output table (TIOT) and of the command scheduler communications list in the command scheduler control block (CSCB). (The CSCB is in the system queue space.) These addresses are the only information provided in response to an EXTRACT macro instruction when using an operating system with MFT without subtasking.

IRB

- The address of the end-of-task exit routine to be given control after the specified task is terminated.

You must provide an area into which the control program places the information you request. If you request the fields GRS, FRS, AETX, PRI, CMC, and TIOT by coding FIELDS=ALL, the area must be seven fullwords long. If you request only a portion of the information, the area must be one fullword in length for each item of information you request. In a system with the time sharing option (TSO) you can also request the fields TSO, PSB, and TJID. If you request information other than the address of the task input/output table when you are using an operating system with MFT without subtasking, each additional item of information requested will result in the corresponding fullword in the answer area being set to zero.

TIMING SERVICES

The timing services available depend on options selected when the operating system was generated. These options are the time option, which provides the ability to request the date and time of day, and the interval option, which includes the time option functions and also provides the ability to set, test, and cancel intervals of time. The interval option is standard in an operating system with MVT; either option can be selected with the other configurations of the operating system. If neither of these options was selected, the date is the only timing service provided. In the Model 65 Multiprocessing system, timing services must only be obtained through the use of the supervisor macro instructions: STIMER, TIME, TTIMER. Direct reference to the interval timer location in a multiprocessing system may produce unpredictable results.

DATE AND TIME OF DAY

The operator is responsible for initially supplying the correct date and time of day information, based on a 24-hour clock, for control

program use. The control program updates the time of day information every 16.7 milliseconds for 60 cycle-per-second line frequency, or every 20 milliseconds for 50 cycle-per-second line frequency. You request the date and time of day information using the TIME macro instruction. The control program returns the date in register 1 and the time of day in register 0.

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 day of the year is shown in Example 22.

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 micro-seconds.) 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 both the time and interval options have not been selected, the operand is ignored and the content of register 0 is set to zero.

TIMING SERVICES ON THE IBM SYSTEM/370

In an MFT or MVT system generated for System/370 all references to time of day and date use the time-of-day (TOD) clock. The TOD clock, a feature of System/370, is a 64-bit binary counter. (For more information about the TOD clock, see IBM System/370 Principles of Operation.) Bit 51 of the counter is equivalent to one microsecond.

The TOD clock is incremented continuously while the power is on; the clock is not affected by the system stop conditions that affect the interval timer in location 80. 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.

DATE AND TIME OF DAY

If you use the TIME macro instruction with the BIN, TU, and DEC operands, the date is returned in register 1 and the time of day is returned in register 0. With the MIC,address operand, the time of day is returned as an unsigned 64-bit binary number in the area specified by "address." The time of day is returned with bit 51 equivalent to one microsecond. With the MIC,address operand, register 0 is set to zero.

```
         ...
         TIME              ,  Request date
         ST    1,ANS          Store packed date
         UNPK  DOUBLE,ANS     Unpack date for printing
         ...
ANS      DS    F              Fullword for packed date
DOUBLE   DS    D              Double word for unpacked date
```

Example 22. Day of Year Processing

INTERVAL TIMING

A time interval 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. When you are using an operating system with MFT without subtasking, only one time interval can be in effect at any one time during the job step. With an operating system with MVT or MFT with subtasking, each task in the job step can have an active time interval.

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

- BINTVL - requires an unsigned 32-bit binary number, the low order bit having a value of 0.01 second.
- 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.

When you test the time remaining in the interval, 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 the interval has already expired, the content of register 0 is set to zero.

When you request a time interval, you also specify the manner in which the interval is to be decremented, 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 decremented continuously whether the associated task is active or not. TASK indicates that the interval is to be decremented 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 main 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.

Example 23 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, to be decremented 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 completes, the TTIMER macro instruction causes any time remaining in the interval to be canceled; the exit routine is not given control. If, however, the loop is still in effect when the time interval expires, control is given to the exit routine FIXUP. The exit routine saves registers and turns on

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

Example 23. Interval Timing

the switch tested in the loop. The FIXUP routine could also print out a message indicating that the loop did not complete successfully. Registers are restored and control is returned to the control program. The control program returns control to the main program and processing continues. When the switch is tested this time, the branch is taken out of the loop.

If issued by a timer completion exit routine, a STIMER macro instruction acts as a NOP instruction only for MFT. An exit routine therefore cannot be used to set a new time interval for MFT.

If issued by a timer completion exit routine, a STIMER macro instruction is honored for MVT. However, the STIMER issued from the exit routine should not specify that same exit routine. If it does specify the same exit routine, an infinite loop will occur.

The accuracy of a time interval is affected by two factors: the resolution of the timer and the "competition" of other tasks for control. The resolution of the timer (the time between successive updating of the timer) is 16.7 milliseconds for 60 cycle per second line frequency. An attempt to measure an interval of less than 16.7 milliseconds or an attempt to time to an accuracy of greater than 16.7 milliseconds can lead to erroneous results.

When you are using an operating system with MFT or MVT, 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 decremented 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 control 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.

WRITING TO ONE OR MORE OPERATOR CONSOLES

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. When an MFT, MVT, or Model 65

Multiprocessing operating system has the Multiple Console Support (MCS) option, messages can be sent to (and replies can be received from) as many as 32 operator consoles.

To use the WTO macro instruction, you code your 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.

Routing of the message (in a system with the MCS option) is performed 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 the appendix of the Supervisor and Data Management Macro Instructions publication.

Disposition of the message (in a system with the MCS option) is indicated through the descriptor codes specified in the WTO macro instruction. Descriptor codes functionally classify WTO messages so that they may be properly presented on, and deleted from, display type 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 one or two is coded into the WTO macro instruction, an asterisk (*) is inserted as the first character of the message. The asterisk informs the operator that he is required to take some immediate action. If a descriptor code other than one or two 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 the appendix of the Supervisor and Data Management Macro Instructions book.

A sample WTO macro instruction is shown in Example 24. The routing code (ROUTCDE) and descriptor code (DESC) keyword parameters are ignored if the operating system does not have the MCS option.

To use the WTOR macro instruction, you code the message exactly as designated in the WTO macro instruction. 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 asterisk as the first character of all WTOR messages, thereby informing the operator that immediate action is required. You must, however, indicate the operator 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. You also supply the address of an event control block which the control program will post after the reply has been placed, left-adjusted, in your designated area. (The use of the event control block is discussed under the heading "Task Management.")

A sample WTOR macro instruction is shown in Example 25. The routing code and descriptor code values are ignored if the operating system does

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

Example 24. 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
```

Example 25. Writing to the Operator With a Reply

not have the MCS option. In a system with MFT or MVT, 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 to more than one functional area (where the WTOR has more than one routing code), any console within those areas has the authority to reply. The first reply received by the operating system 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 operating system moves the reply into the issuer's reply area and posts the event control block as completed. 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 nct 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 (SYSGEN) time, your installation determines how many 176-byte system message blocks (SMBs) to allow. You can override this number at initial program load (IPL) time; 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 14.

WRITING TO THE HARD COPY LOG

When using an operating system that has the Multiple Console Support (MCS) 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.

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 messages 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 14. Using WTO and WTOR to Write Messages to the Programmer

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

Operating systems with MFT, MVT, or Model 65 Multiprocessing provide a system log as an optional feature. The system log 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 system, user programs, and/or 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 currently open and receiving input, is logically connected to two buffers. The operating system 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 system 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 system 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.
- Tne 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 WTL 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 the Model 85 Operator Console with cathode ray tube (CRT) display as a console, unnecessary messages can be deleted from the operator's screen by the programmer.

The operating system assigns a message identification number to each WTO and WTOR message, and returns the message to the program 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.

PROGRAM INTERRUPTION PROCESSING

Unusual conditions encountered in a program cause a program interruption. These conditions include incorrect operands and operand specifications, as well as exceptional results, and are know 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 any program interruptions occur, and 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 exception. 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. 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 results in a control program parameter list, called a program interruption control area (PICA). The PICA, shown in Figure 15, 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 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.

Example 26 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 main storage area assigned to the job

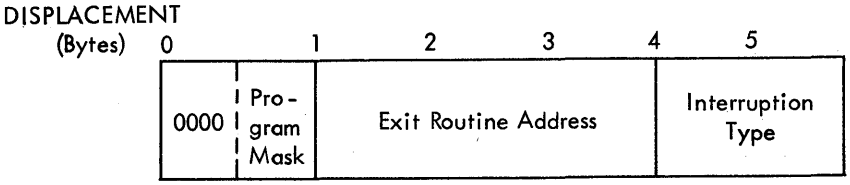


Figure 15. Program Interruption Control Area

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

Example 26. Use of the SPIE Macro Instruction

step (subpool 0 in an operating system with MVT). 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 16.

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

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

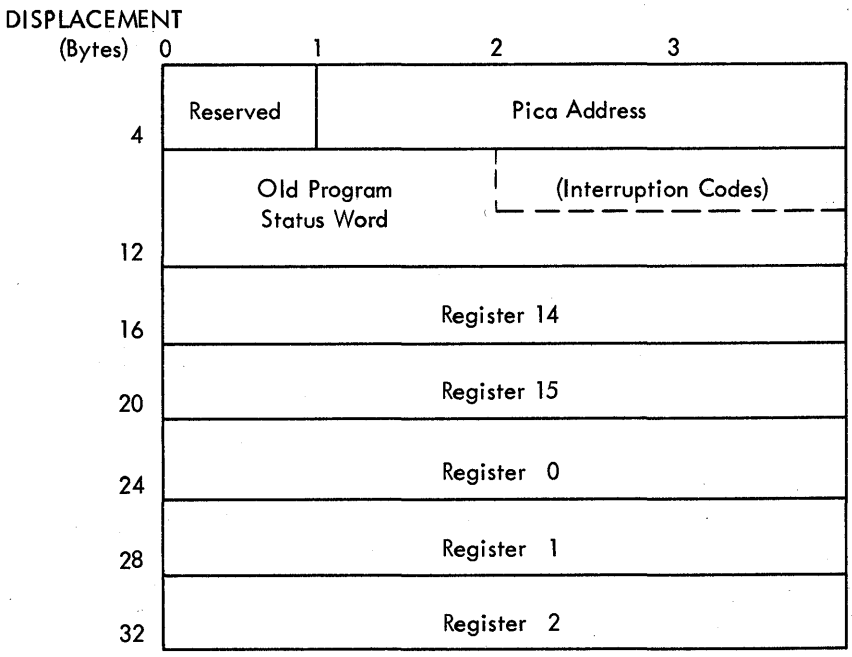


Figure 16. Program Interruption Element

- Register 0: internal control program information.
- Register 1: address of the program interruption element for the task that caused the interruption.
- Registers 2-12: same as when the program interruption occurred.
- Register 13: address of the save area for the main program. The exit routine must not use this save area.
- Register 14: return address (to the control program).
- Register 15: address of the exit routine.

The exit routine must be in main 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 interrogate 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. This is 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.

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

PRECISE AND IMPRECISE INTERRUPTIONS

After an interruption, the old program status word contains the address of the next instruction to be executed in bits 40-63, and the length of the previous instruction in bits 32 and 33. In System/360 Models 65, 67, 75, 85, 91, 195, and System/370 Model 165, however, the address of the next instruction may not be precise; if the address is not precise, the instruction length code (ILC) in bits 32-33 is set to zero. You should therefore test the instruction length code for zero before using the next instruction address.

In Models 65-85, imprecise interruptions can result only from protection and addressing exceptions. In the Model 91, imprecise interruptions result from these and eight other types of exceptions. In the Model 195, imprecise interruptions result from nine other types of exceptions. Figure 17 summarizes the types of program exceptions that can result in an imprecise interruption.

Except for the protection exception in the Model 91, any exception that can result in an imprecise interruption can also result in a precise interruption. You therefore should not assume that a specific

type of exception will always produce an imprecise interruption. Figure 17 defines the conditions under which interruptions are precise in Models 65-195. Note that interruptions are always precise in systems with lower model numbers.

INTERRUPTIONS IN THE MODELS 91 AND 195

As shown in Figure 18, the interruption code in the Models 91 and 195 differs for precise and imprecise interruptions. For precise interruptions (as for all interruptions in other models), exceptions are indicated in bits 28-31 of the old program status word. For imprecise interruptions, bits 28-31 are zero, and exceptions are indicated in bits 16-27.

Before testing the interruption code to determine the cause of an interruption, you should test the instruction length code to determine whether the interruption is precise or imprecise. If the instruction length code is zero, indicating an imprecise interruption, you should test bits 28-31 of the old program status word to determine whether the interruption has occurred on a Model 91 or 195. If bits 28-31 are zero, the interruption has occurred on a Model 91 or 195 and the cause of the interruption is indicated in bits 16-27. If bits 28-31 are not zero, the interruption has not occurred on a Model 91 or 195, and these bits themselves indicate the cause of the interruption.

Type of Exception	Type of Interruption							
	Precise (ILC≠0)		Imprecise (ILC=0)					
	All Models		Models 65-85 and System/370 Model 165		Model 91		Model 195	
	Bits 16-27	28-31	Bits 16-27	28-31	Bits 16-27	28-31	Bits 16-27	28-31
Operation	(zero)	0001						
Privileged Operation	(zero)	0010						
Execute	(zero)	0011						
Protection	(zero)	0100	(zero)	0100	100000000000	(zero)	100000000000	(zero)
Addressing	(zero)	0101	(zero)[1]	0101[1]	010000000000	(zero)	010000000000	(zero)
Specification	(zero)	0110			001000000000	(zero)		
Data	(zero)	0111			000100000000	(zero)	000100000000	(zero)
Fixed-Point Overflow	(zero)	1000			000010000000	(zero)	000010000000	(zero)
Fixed-Point Divide	(zero)	1001			000001000000	(zero)	000001000000	(zero)
Decimal Overflow	(zero)	1010					000000000010	(zero)
Decimal Divide	(zero)	1011					000000000001	(zero)
Exponent Overflow	(zero)	1100			000000100000	(zero)	000000100000	(zero)
Exponent Underflow	(zero)	1101			000000010000	(zero)	000000010000	(zero)
Siginificance	(zero)	1110			000000001000	(zero)	000000001000	(zero)
Floating-point Divide	(zero)	1111			000000000100	(zero)	000000000100	(zero)
			[1]Except Model 65					

Figure 17. Interruption Code in the Old Program Status Word

Type of Exception	Models 65-85 and System/370 Model 165		Model 91				Model 195		
	Always Precise	Sometimes Precise[1]	Always Precise	Sometimes Precise[2]	Precise in INHIBIT OVERLAP Mode[3]	Precise for Decimal Simulation[4]	Always Precise	Sometimes Precise[5]	Precise in INHIBIT OVERLAP Mode[3]
Operation	X		X				X		
Privileged Operation	X		X				X		
Execute	X		X				X		
Protection		X				X			
Addressing		X		X		X		X	
Specification	X			X		X	X		
Data	X				X	X			X
Fixed-point Overflow	X				X				X
Fixed-point Divide	X				X				X
Decimal Overflow	X					X			X
Decimal Divide	X					X			X
Exponent Overflow	X				X				X
Exponent Underflow	X				X				X
Significance	X				X				X
Floating-point Divide	X				X				X

[1]A protection or addressing exception results in a precise or imprecise interruption, depending on the cause of the exception.
[2]An addressing or specification exception results in a precise or imprecise interruption, depending on the cause of the exception. For details, refer to the Model 91 Functional Characteristics publication.
[3]The indicated interruptions are precise if the INHIBIT OVERLAP switch is set on the system control panel.
[4]The interruption for a protection exception is precise only when simulated by the control program decimal simulator routine. Interruptions for decimal overflow and decimal divide exceptions occur only as simulated interruptions; they do not occur if the control program does not include the decimal simulator routine.
[5]An addressing exception results in a precise or imprecise interruption, depending on the cause of the exception. For details, refer to the Model 195 Functional Characteristics publication.

Figure 18. Precise Interruptions in IBM System/360 Models 65, 67, 75, 85, 91, 195, and System/370 Model 165

In the Model 91, there are ten types of program exceptions that can cause an imprecise interruption. In the Model 195, there are eleven types of program exceptions that can cause an imprecise interruption. Each is represented by a separate bit in the interruption code (bits 16-27). After an imprecise interruption, the interruption code may indicate more than one type of exception. When it does, the indicated exceptions may be due to a single instruction, or to several instructions whose execution was overlapped. Note that each of the indicated exceptions may have occurred more than once, and there is no indication as to which occurred first.

If you provide an exit routine to handle any of the exceptions that may result in an imprecise interruption, you should specify all ten such exceptions in the SPIE macro instruction. When an imprecise interruption occurs, your exit routine will be entered only if the PICA indicates all of the exceptions that are indicated in the old program status word. For example, if you provide a routine to handle fixed-point overflow, and if you specify only fixed-point overflow in the SPIE macro instruction, the routine will not be entered if both fixed-point overflow and specification exceptions are indicated for the same interruption.

DECIMAL SIMULATION IN THE MODEL 91

The instruction set for the model 91 does not include the decimal instructions AP, CP, DP, MP, SP, and ZAP; each of these instructions causes an operation exception, which results in a precise interruption. If the decimal simulator routine was specified at system generation, the control program simulates the decimal operation. Otherwise, control is passed to your program interruption exit routine, or to the control program exit routine.

Decimal simulation may result in an exceptional condition. When it does, the control program simulates a precise interruption as indicated in Figure 18. For decimal overflow, execution is completed and the condition code is set. For other exceptions, execution is suppressed; the condition code and the contents of main storage remain unchanged. Note that the control program does not simulate an interruption for decimal overflow if the interruption is disabled.

EXTENDED-PRECISION FLOATING-POINT SIMULATION

The OS/360 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 Supervisor and Data Management Macro Instructions and the formats of the other extended-precision floating-point instructions are described in the Principles of Operation.

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 above under "Program Interruption Processing."

If you wish 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:

- Prepare a parameter list to pass to IEAXPSIM.
- Pass control to IEAXPSIM, using standard operating system conventions.

- Prepare a parameter list to pass to the simulator.
- Pass control to the simulator, using standard operating system conventions.
- Check the code returned by the simulator.
- Perform corrective action if necessary.

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

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

Read the following paragraphs and then look at Example 27, which 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 or 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.

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 Example 27 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 Example 27, the doubleword is SIMUL.

Before passing control to the simulator, you can use the LOAD macro instruction to bring the simulator into main 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 19.

```
         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 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                 Word area
SIMSV    DS      16F                 Register area
SIMADD   DC      F'0'                Address of simulator
RET      DS      F                   Return address
SAVESIM  DS      18F                 Save area
```

Example 27. Calling the Extended-Precision Floating-Point Simulator

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

Figure 19. Return Codes from the Extended-Precision Floating-Point Simulator

If the return code was X'FF', the exit routine determines the kind of error encountered by the simulator by examining the interruption code in bits 28-31 of the PSW. Figure 20 shows the possible settings of the interruption code.

The simulator will adjust 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-interrupt 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.

Meaning of Interruption	Bits 28-31
The simulator found that the operation was not an extended-precision floating-point operation and returned control without further processing.	0001
Protection exception [1] [3]	0100
Addressing exception [1] [3]	0101
Specification exception [1] [2] [3]	0110
Exponent overflow exception [4]	1100
Exponent underflow exception [4]	1101
Significance exception [4]	1110
Floating-point divide [4]	1111

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

Figure 20. Interruption Codes Returned by the Simulator

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

To use the simulator, you need to code a SPIE macro instruction, such as the one below which specifies the exit routine named EXTPRE as the one to be given control if an operation interrupt occurs.

```
...
SPIE     EXTPRE,(1)
...
```

The routine EXTPRE sets up a parameter list and calls the extended-precision floating-point simulator. When control is returned from the simulator, tests are made for errors and exceptional conditions. The rest of what you have to do to use the simulator is shown in Example 27.

If the full simulator (IEAXPALL) is loaded on a CPU that already has the extended-precision floating-point facility, no abnormal conditions will result. Only the DXR macro instruction will be 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 will occur 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.

ABNORMAL CONDITION HANDLING

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 21 is an example of one of these. The routine shown in Figure 21 checks a control field in an input parameter list to determine which function the program is to perform. Only characters between 1 and 4 are valid in the control field. The presence of any other character is invalid, but the routine must be prepared to detect and handle these characters. The routine should indicate its inability to continue processing by returning control to the calling program with an error return code. The calling program should then try to interpret the return code and to recover from

the error. If it cannot do so, the calling program should detach its incomplete subtasks, execute its usual termination procedures, and return control to its calling program, again with an error return code. This procedure may result in termination of all the tasks of a job step; if it does, the COND parameters of the JOB and EXEC statements may be used to determine whether or not 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 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.

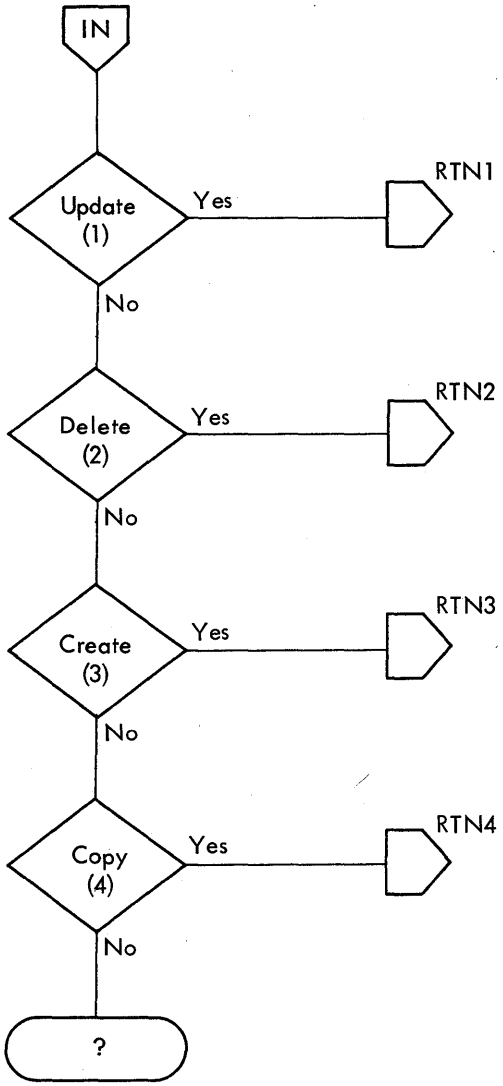


Figure 21. Abnormal Condition Detection

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 supervisor to abnormally terminate the job step task. The most frequent cause of this is that the subtask does not have sufficient main storage for ABEND's processing. ABEND "steals" main 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 main storage during the performance of the task.
- Releases the main 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 which 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 which was active when the ABEND macro instruction was issued. A DETACH macro instruction is issued by the control program for each of the subtasks.
- 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 designated in the ATTACH macro instruction issued to create the active task, the completion code specified in the ABEND macro instruction is placed in the designated event control block, and the completion bit is turned on.
- If the ETXR operand was designated 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 designated 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 main 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. In

In any operating system, 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 later in this section.

INTERCEPTING ABNORMAL TERMINATION OF TASKS

Abnormal termination of a task can be intercepted through the use of the STAE macro instruction. When a task that has previously issued a STAE macro instruction is scheduled for abnormal termination, termination processing is intercepted and control is returned to the user at his STAE exit routine address, as specified in the STAE macro instruction. Within the STAE exit routine, the user can perform pre-termination functions or diagnose the error. He can also determine whether abnormal termination should continue for the task, or whether a STAE retry routine, which would circumvent abnormal termination, should be scheduled. For further information on scheduling a STAE retry routine, see the System Programmer's Guide.

At the time the abnormal termination is scheduled, the STAE exit routine must be resident. It must either be part of the program issuing STAE or be brought into storage via the LOAD macro instruction.

The STAE exit routine can contain an ABEND macro instruction, but it must not contain a STAE or an ATTACH macro instruction.

A single user program can issue more than one STAE macro instruction with the create (CT) operand. Each issuance makes the previous STAE environment temporarily inactive. The suspended STAE environment can be reestablished by canceling the current STAE. Unless it is intended that the existing STAE environment be saved, it should be canceled prior to issuing another STAE. Otherwise, main storage will be wasted by STAE control blocks for inactive STAE environments.

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

The user can cancel (make the previous STAE request active) or overlay the current STAE request. The STAE request that is canceled or overlaid is the one most recently made. If no STAE requests are active for the task at the time a cancel or overlay is issued, or if the user attempts to cancel or overlay a STAE request not associated with his Request Block level of control, he will be informed that his request is invalid by a return code. A STAE request can be canceled by issuing the STAE macro instruction with the STAE exit routine address specified as

```
         ...
         STAE  EXIT1,CT,PARAM=LIST1,                                  C
               XCTL=YES,ASYNCH=YES,                                   C
               PURGE=QUIESCE       Initial STAE request
         ...
         ...
         LA    5,EXIT2             Put new exit routine address in
                                     register 5
         STAE  (5),OV,PURGE=NONE   STAE request for overlay
         ...
LIST1    DC    F'0'                Parameter list for exit routines
         DC    X'A0'
EXIT1    EQU   *                   Entry point of first exit routine
EXIT2    EQU   *                   Entry point of second exit routine
```

Example 28. Use of STAE Macro Instruction

zero. Overlaying is done by issuing a STAE macro instruction specifying OV.

When a program issuing STAE returns control to a previous level via an SVC 3, all STAE requests issued by that program are canceled. A STAE request specifying XCTL=YES is not canceled when the STAE user issues an XCTL macro instruction and the STAE environment is connected to the program in control after XCTL. If a program terminates by any means other than an SVC 3 or a RETURN macro instruction, all STAE requests must be canceled by the terminating program before returning control to another program.

STAE requests issued by a program are queued for that program so that the last STAE request issued is the active one, that is, it is the one that causes the STAE exit routine to receive control if the program is abnormally terminated. If the active STAE request is canceled, the next-to-the-last STAE request becomes the last and thus the active one.

Example 28 shows the use of the STAE macro instruction. The STAE request is initially made specifying a STAE exit routine address (EXIT1) and parameter list address (LIST1). The XCTL=YES parameter indicates that this STAE request will not be canceled if the program terminates via the XCTL macro instruction. The ASYNCH=YES parameter indicates that asynchronous interruptions will be allowed during STAE exit routine processing. The PURGE=QUIESCE parameter indicates that input/output requests not yet performed are removed from the system's active input/output queue (purged) but can later be returned to that queue (restored). If PURGE=QUIESCE cannot be honored by the system, the input/output requests are removed from the queue with the halt option and therefore cannot be restored.

In the second issuance of STAE, the previous STAE request is modified through the overlay (OV) option. The STAE exit routine address is now EXIT2, and input/output intervention will now be bypassed, but the parameter list address, the XCTL=YES, and the ASYNCH=YES remain the same.

After a STAE macro instruction has been issued, the register contents upon return to the user are as follows:

- Registers 0, 1: Unpredictable.
- Registers 2-13: Same as when STAE was issued.
- Register 14: Unpredictable.

- Register 15: Completion code.

Decimal Code	Indication
0	Successful completion of creating, overlaying, or canceling a STAE request.
4	No storage obtainable for a STAE request.
8	A STAE request to be canceled or overlaid did not exist, or a STAE was issued in the user's exit routine.
12	Invalid exit routine or parameter list address.
16	Attempt to cancel or overlay another user's STAE request.

When a program with an active STAE request encounters an ABEND situation, control is passed to the STAE exit routine. ABEND processing continues and the STAE exit routine does not receive control in the following situations:

- If the abnormal termination is caused by an operator's CANCEL, job step timer expiration, or the detaching of an incomplete task.
- If the terminating task is in must complete status and problem program mode. (Putting a task in the must complete status is explained in the System Programmer's Guide.)
- If the OUTLIMIT is exceeded for SYSOUT.
- If an invalid ABEND recursion (an abnormal condition encountered during abnormal termination) occurs.
- If an abnormal condition is encountered during normal termination.
- If the failing task has been in a wait state for more than 30 minutes.
- If the STAE macro instruction was issued by a subtask and the mother task abnormally terminates.
- If the exit routine was specified for a subtask, via the STAI operand of the ATTACH macro instruction, and the mother task abnormally terminates.
- If the abnormal termination is because the task that issued the STAE still has active subtasks when it returns to the control program via an SVC 3.
- If any other problem arises while the control program is preparing to give control to the STAE exit routine.

Before the STAE exit routine receives control, any existing SPIE requests are canceled and the purge request specified in the STAE macro instruction is fulfilled. The register contents upon entry to the STAE exit routine are as follows:

- Register 0:

Decimal Code	Indication
0	Active I/O at the time of the ABEND was quiesced and is restorable.

4 — Active I/O at the time of the ABEND was halted and is not restorable.

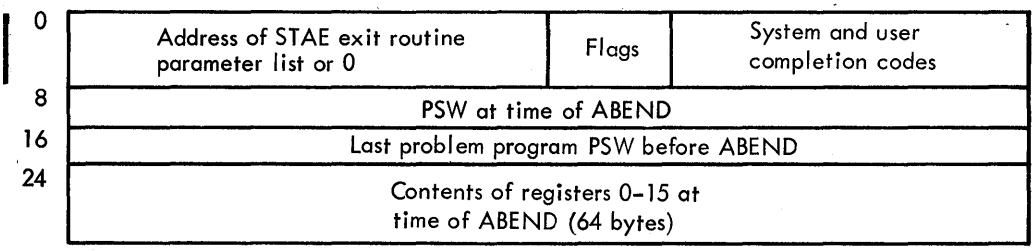
8 — No I/O was active at the time of the ABEND.

- Register 1: Address of a 104-byte work area, as shown in Figure 22.
- Registers 2-12: Unpredictable.
- Register 13: Address of a supervisor-provided register save area.
- Register 14: Return address.
- Register 15: Address of the STAE exit routine.

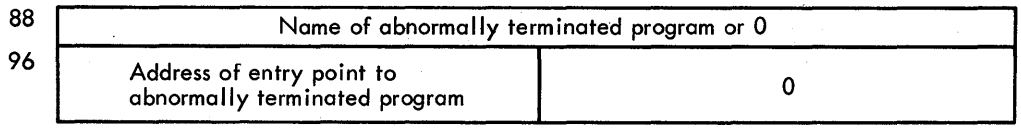
Note: Registers 13 and 14, if used by the STAE exit routine, must be saved and restored prior to returning to the calling program. Standard subroutine linkage conventions apply.

Bytes 4-7 in Figure 22 are used as follows:

Bit	Content	Indication
0	1	Dump to be given.
0	0	Dump not to be given.
1	1	Job step to be terminated.
1	0	Only failing task to be terminated.
2-7	--	Not used.
8-19	--	System completion code (packed, unsigned, decimal).
20-31	--	User completion code (hexadecimal).



If a problem program issued STAE:



If supervisor program issued STAE:

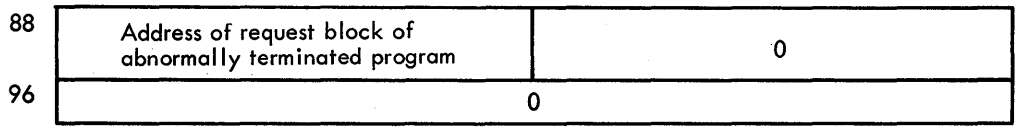


Figure 22. Work Area for STAE Exit Routine

If main storage was not available for the work area, the register contents upon entry to the STAE exit routine are as follows:

- Register 0: 12 (decimal)
- Register 1: Flags and completion codes (see Figure 22, bytes 4-7 for format).
- Register 2: Address of STAE exit parameter list.
- Register 3-13: Unpredictable.
- Register 14: Return address.
- Register 15: Exit routine address.

Note: If a work area could not be provided by the control program, a register save area will not be provided either. A save area is never provided for a retry routine.

Before returning control to the operating system from the STAE exit routine, the user must put a return code in register 15. The return code indicates whether ABEND processing is to be continued for the task or whether a STAE retry routine should be scheduled. (The details about scheduling a STAE retry routine are in the System Programmer's Guide.)

The return codes to be placed in register 15 are defined as follows:

Decimal Code	Indication
0	No retry routine provided.
4	A STAE retry routine has been provided and the Request Block chain should be purged.
8	A STAE retry routine has been provided and the Request Block chain should not be purged. (To be used by routines in supervisor state only.)
12	A STAI (Subtask ABEND intercept) retry routine has been provided.
16	No further STAI processing; ABEND processing is to continue.

For further information on the option of STAE retry, see the System Programmer's Guide.

INTERCEPTING ABNORMAL TERMINATION OF SUBTASKS

To provide an exit in your program to intercept abnormal termination of a subtask, use the STAI (subtask ABEND intercept) operand of the ATTACH macro instruction you issue to create the subtask. The STAI request issued for any subtask will be propagated for all subtasks further down the tree. For example, Task A attaches Task B and uses the STAI operand on the ATTACH macro instruction. When Task B attaches Task C, the STAI request issued by A will be active for C as well as B.

Since more than one subtask may abnormally terminate at the same time, the STAI exit routine may be used by more than one task concurrently. Therefore, the exit routine must be reenterable, or it may fail during the second entry.

THE DUMP

There are three types of main-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.
- A core image 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 core image 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 generally requests 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 access technique using sequential organization (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 is 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 Programmer's Guide to Debugging. The entire dump shown in that publication is provided in an abnormal termination dump if a DD statement with a ddname 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 ddname 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 4) 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 ddname must be SYSABEND or SYSUDUMP; for a SNAP macro instruction dump, the ddname must be any name except SYSABEND or SYSUDUMP. The requirements for writing the DD statement are described in the Programmer's Guide to Debugging.

- 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=nnn, and LRECL=125, where nnn is 882 for MFT and either 882 or 1632 for MVT. (The data control block is discussed in the Data Management Services manual.) If your program is to be processed by the loader, you should also issue a CLOSE macro instruction for the SNAP data control block.

- Sufficient unused main storage must be available in the area assigned to the job step to hold the control program dump routine and, if not already in main storage, the BSAM data management routines. For an abnormal termination dump, additional main storage is required for the routines to process the OPEN macro instruction issued by the control program, and for the trace table. Refer to the Storage Estimates publication for storage requirements.

INDICATIVE DUMP

In an operating system with MFT, you can obtain an indicative dump, as shown in the Programmer's Guide to Debugging. 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 ddname SYSABEND or SYSUDUMP, or the control program entry for that DD statement was destroyed. The indicative dump is printed on the system output device. The indicative dump is not provided in an operating system with MVT.

CORE IMAGE DUMP

If a system routine fails, the system automatically supplies a dump of main storage. This dump, called the core image dump, provides diagnostic information. The system writes the core image dump in the system data set SYS1.DUMP or in a tape volume at the device designated when the operating system was initially loaded.

In systems with MFT or MVT, use the IMDPRDMP Service Aid program to obtain a printout of the dump. A description of IMDPRDMP and the core image dump formats appear in the Service Aids publication.

For guidance in using the core image dumps from all configurations of the operating system, refer to the Programmer's Guide to Debugging.

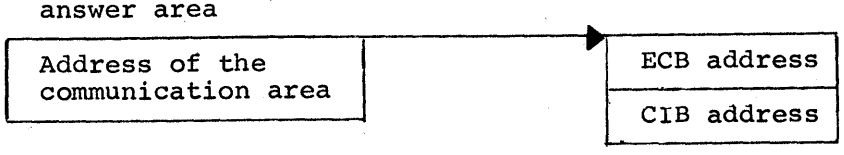
OPERATOR COMMUNICATION WITH A PROBLEM PROGRAM

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

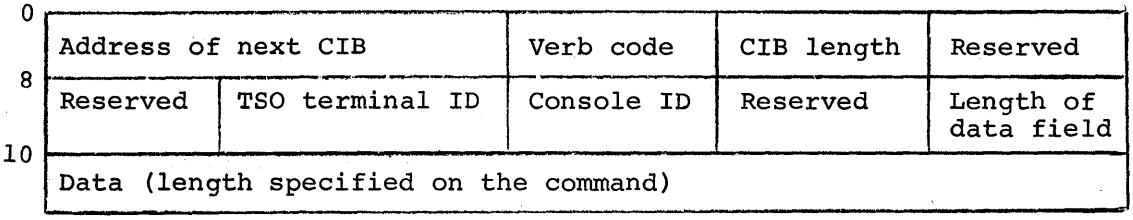
An EXTRACT macro instruction is issued to obtain a pointer to the communications ECB, which is posted when a STOP or a MODIFY command is issued, and a pointer to the first Command Input Buffer (CIB) on the CIB chain for the task:

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

EXTRACT will return the following:



The CIB contains the information specified on the STOP or the MODIFY command:



Verb code x'04' START
x'40' STOP
x'44' MODIFY

If the job was started from the console, the CIB pointed to when the EXTRACT macro instruction is issued will be the START CIB. If the job was not started from the console, the address of the first CIB will be zero. If the address of the START CIB is present, the QEDIT macro instruction should be used to free this CIB after any parameters passed in the START command have been examined:

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

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

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

The value of n is any integer value from 0 to 255. If n is set to zero, no MODIFY commands will be accepted for the job. STOP commands, however, will be accepted for the job regardless of the value set for CIBCTR.

For the duration of the job, the communications ECB may be waited on or checked at any time to see if a command has been entered for the program. The verb code in the CIB should be examined to determine whether a STOP or a MODIFY command has been entered. After the data in the CIB has been processed, a QEDIT macro instruction should be issued to free the CIB.

The communications ECB will be cleared each time a CIB is freed. Care should be taken if multiple subtasks are examining these fields. Any CIBs not freed by the task will be unchained by the system when the task is terminated. The area addressed by the pointer obtained by the EXTRACT macro instruction, the communications ECB, and all CIBs are in protected main storage and may not be altered.

No matter which configuration of the operating system you are using, there is a finite amount of main storage available to your job step. you are using the primary control program, you have available all main storage not used by the control program; if you are using an operating system with MFT or MVT, you have a partition or region of fixed size available to your job step.

In an operating system with MFT, the main storage available to problem programs is divided into 1 to 15 fixed partitions. The division is made during system generation, but the operator can enlarge a partition by combining it with others. Each partition is associated with one or more "job classes," which can be varied by the operator. On the basis of job class and priority specified in a JOB statement, a job is assigned to a partition and scheduled for execution. A job step will be abnormally terminated if it requires more main storage than is available in the partition.

In a system with MVT, available main storage is divided into regions, which vary in size and number according to the requirements of the job steps being performed. Job steps are selected for execution according to job class and priority, and each is assigned a region of the size specified in a JOB or EXEC statement. If the highest priority job step requires a larger region than can be made available, its execution is delayed, and a lower priority job step (one with sufficiently lower storage requirements) is initiated. After a job step has been initiated, its region can be extended only if the rollout/rollin option has been included in the system. (For a description of rollout/rollin, refer to the System Programmer's Guide.)

You obtain the use of the main storage area assigned to your job step through implicit and explicit requests for main storage. The use of a LINK macro instruction is an implicit request for main storage; the control program allocates space 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 main storage to be allocated to the active task. In addition to your requests for main 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.

The following paragraphs discuss some of the techniques that can be applied for efficient use of the main storage area reserved for your job step. These techniques apply as well to the data management portions of your programs. The specific data management main storage allocation facilities are discussed in Section II of this publication; the principles discussed here provide the background you will need to use these facilities.

EXPLICIT REQUESTS

Main storage can be explicitly requested for the use of the active task by issuing a GETMAIN macro instruction. The main storage request is satisfied by allocating a portion of the main storage area reserved for the job step to the active task. You cannot use the main storage area reserved for the job step without first requesting it; if you attempt to use it without requesting it, the task is abnormally terminated. The main storage area is not set to zero when allocated.

You return control of main storage by issuing a FREEMAIN macro instruction. This does not release the area from control of the job step; it only makes the area available to satisfy the requirements of additional requests for any task in the job step. The main storage assigned to a task is also released for other uses when the task terminates, except as indicated under "Subpool Handling."

SPECIFYING LENGTHS

Main storage areas are always allocated to the task in multiples of eight bytes and begin on a doubleword boundary. The request for main 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 area and, because you only make one request, the amount of control program overhead is less.

TYPES OF EXPLICIT REQUESTS

There are four methods of explicitly requesting main 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 three methods do not produce reenterable code unless coded in the list and execute forms as indicated in the paragraph "Implicit Requests." The methods are as follows:

REGISTER TYPE (R): Specifies a request for a single area of main storage of a specified length. The address of the area is returned in register 1. This type of request produces reenterable code, 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 main storage of a specified length. The control program places the address of the allocated area in a fullword you supply.

LIST TYPE (L): Specifies a request for one or more areas of main storage. You place the length of each area in a list; each list entry represents a request for one area of main storage. The control program places the addresses of the allocated areas in consecutive full words in another list you supply. The addresses are placed in the list in the same order they were requested. This type of request can be made only in an operating system with MVT.

VARIABLE TYPE (V): Specifies a request for a single area of main storage with a length between two values you specify. The control program will attempt 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 you supply.

In addition to the above methods of requesting main storage, you can designate the request as conditional or unconditional. (A register type request is always unconditional.) If the request is unconditional and sufficient main storage is not available to fill the request, the activetask is abnormally terminated. If the request is conditional, however, and insufficient main storage is available, a return code of four is provided in register 15; a return code of zero is provided if

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

Example 29. Use of the GETMAIN Macro Instruction

the request was satisfied. When a conditional list-type request is made, no main storage is allocated unless all of the requested areas can be allocated.

An example of the use of the GETMAIN macro instruction is shown in Example 29. The example assumes a program which operates most efficiently with a work area of 16,000 bytes, with a fair degree of efficiency with 8000 bytes or more, inefficiently with 4000 to 8000 bytes, and not at all with less than 4000 bytes. The program uses a reenterable load module with an entry point 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 would be available when it was required.

A conditional request for a single element of main storage with a length of 16000 bytes is requested in Example 29. The return code in register 15 is tested to determine if the area was available; if the return code was zero (the 16,000 bytes were allocated), control is passed to the processing routine. If sufficient area was not available, an attempt to obtain more main storage area 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 4000 and 16000 bytes in length. If the minimum size is not available, the task is abnormally terminated. If at least 4000 bytes were available, however, 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.

SUBPOOL HANDLING (IN MFT SYSTEMS WITHOUT SUBTASKING)

There is only one unnumbered subpool in an operating system with MFT. In this configuration of the operating system all main storage requests

are satisfied by allocating storage from this unnumbered subpool. If subpool numbers are specified, the numbers are ignored if they are not greater than 127 (the greatest number that is valid in a system with MVT). If subpool numbers greater than 127 are specified, the job step is abnormally terminated.

SUBPOOL HANDLING (IN MFT SYSTEMS WITH SUBTASKING)

Although subpools are not created in MFT systems, it is convenient to call the partition itself "subpool 0." That is, all main storage in a partition is shared by all tasks active in that partition. Main storage not allocated to any task is called "free storage." "Subpool 240" is used by the supervisor to enable the sharing of a reenterable program invoked by a LOAD macro instruction. "Subpool 255" is used by the supervisor to request storage from the system queue area. User programs may request main storage from the partition by specifying any subpool number from 0 to 127 or by specifying no number at all (this provides compatibility with MVT). User-program implied requests for storage, initiated when the user executes an ATTACH, LINK, LOAD, or XCTL macro instruction, are recorded by the supervisor in order for the storage to be freed during termination.

SUBPOOL HANDLING (IN MVT SYSTEMS)

In an operating system with MVT, subpools of main storage are provided to assist in main 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 main storage, a discussion of main storage control is presented here.

MAIN STORAGE CONTROL

When the job step is given a region of main 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 main 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 main storage is allocated to a subtask by the user program while the system is executing in the supervisor state or with a protection key of 0, no other task should free that main storage. If some other task does free that main storage, you get unpredictable results.

For purposes of control and main storage protection, the control program considers all main storage within the region in terms of 2048-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 main storage are made. When there is sufficient unallocated main storage within any block assigned to the designated subpool to fill a request, the main storage is allocated to the active task from that block. If there is insufficient unallocated main 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 23 is a simplified view of a main-storage region containing four 2048-byte blocks of storage. All the requests are for main storage

from subpool 0. The first request from some task in the job step is for 504 bytes; the request is satisfied from the block shown as BLOCK A in the figure. The second request, for 2000 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 2000 bytes from BLOCK B to the active task. A third request is then received, this time for 1000 bytes. There is not sufficient unallocated area remaining in BLOCK B (blocks are checked in the order last in, first out), but there is enough space in BLOCK A, so an additional 1000 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 2048-byte block. Request 4, for 3000 bytes, requires that the control program allocate the area from 2 contiguous blocks which were previously unassigned, BLOCK D and BLOCK C. These blocks are assigned to subpool 0.

As indicated in the preceding example, it is possible for one 2048-byte block in subpool 0 to contain many small areas allocated to many different tasks in the job step, and it is possible that numerous blocks could be split up in this manner. Areas acquired by a task other than the job step task are not released automatically on task termination. Even if FREEMAIN macro instructions were issued for each of the small areas before a task terminated, the probable result would be that many small unused areas would exist within each block, while the control program would be continually assigning new blocks to satisfy new requests. To avoid this situation, you can define subpools for exclusive use by individual tasks.

Any subpool can be used exclusively by a single task or shared by several tasks. Each time that you create a task, you can specify which subpools are to be shared. Unlike other subpools, subpool 0 is shared by a task and its subtask, unless you specify otherwise. When subpool 0 is not shared, the control program creates a new subpool 0 for use by the subtask. As a result, both the task and its subtask can request storage from subpool 0, but both will not receive storage from the same 2048-byte block. When the subtask terminates, its main storage areas in

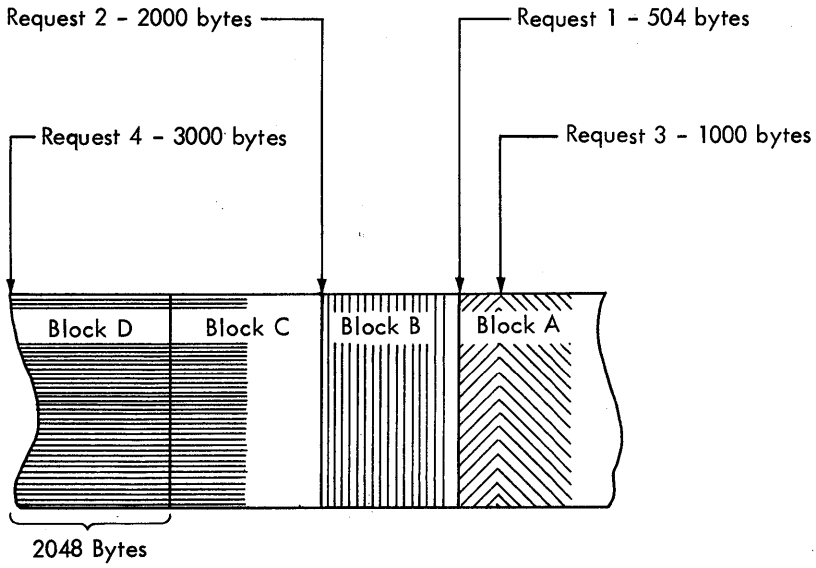


Figure 23. Main-Storage Control

subpool 0 are released; since no other tasks share this subpool, complete 2048-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 2048-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 2048-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 2048-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 sharing the 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 main 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 main storage management is that, by assigning separate subpools to separate subtasks, the breakdown of main 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 main 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

You make an implicit request for main storage every time you issue a LINK, LOAD, ATTACH, or XCTL macro instruction. In addition, you make an implicit request for main storage when you issue an OPEN macro instruction for a data set. The data management routines required to process the data set must be in main storage; the main storage areas used as buffers may also be allocated. When you make an implicit request for more main storage than is available, the active task is abnormally terminated. This section discusses some of the techniques you can use to cut down on the amount of main storage required by a job step, and the assistance given you by the control program.

LOAD MODULE MANAGEMENT

The discussion of program structures indicates the advantages and disadvantages of each of the three types of program designs; simple, planned overlay, and dynamic. The program structure you selected was based on the complexity of the program and the execution time considerations. Once you have selected the program structure, you should plan efficient use of the main storage area that will be assigned

to your job step. Note that main storage is assigned in 2048-byte blocks for implicit requests made in an operating system with MVT. The size of your load modules should be planned to take advantage of this method of allocation. The maximum size load module that can be brought into main storage is 524,248 bytes in an operating system with MFT.

REENTERABLE LOAD MODULES

A reenterable load module is designed so that it does not in any way modify itself during execution. It is "read-only". The advantage of a reenterable load module is most apparent in an operating system with MVT; only one copy of the load module is brought into main storage to satisfy the requirements of any number of tasks in a job step. This means that even though there are six tasks in the job step and each task concurrently requires the load module, the only main storage area requirement is for an area large enough to hold one copy of the load module (plus a few bytes for control blocks). The same main storage requirement would apply if the load module were serially reusable; however, the load module could not be used by more than one task at a time.

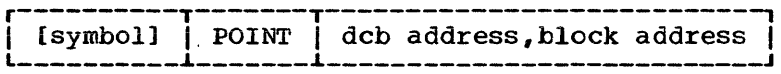
An additional benefit of a reenterable load module occurs when the module is placed in the link pack area. In this case not only is time saved because no loading must be performed, but in addition no main storage area assigned to the job step is required to hold the load module. A link pack area exists only in an operating system with MVT. The contents are established when the operating system is generated and when the operator performs the initial program loading procedure. Any reenterable load module from the link library may be placed in the link pack area. Many of the frequently used data management routines are also placed in the link pack area. If any of your reenterable load modules are used frequently or are used by many jobs, it may save considerable time and space to have those load modules placed in the link pack area.

Because a reenterable module does not modify itself, it offers greater reliability than a nonreenterable module. When there is a machine-check interruption due to a parity error, the Machine-Check Handler program can overlay the damaged copy with a new copy. This is only true for the System/360 Model 65. If the module is designated "refreshable," a fresh copy is loaded automatically by the Machine-Check Handler.

You can designate a module as refreshable without also designating it as reenterable. However, the module must actually be reenterable in its design, because it must not modify itself during execution.

REENTERABLE MACRO INSTRUCTIONS

All of the macro instructions described in the Supervisor and Data Management Macro Instructions manual can be written in reenterable form. From the standpoint of reenterability, 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 use of the macro instructions which pass parameters in registers presents little 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:



One method of coding a reenterable program would be to require that both of these addresses refer to a portion of main 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 general registers 2-12 with the address, and designate the appropriate registers when you code the macro instruction. If register 4 contained the dcb address and register 6 contained the block address, the POINT macro instruction would be written as follows: POINT (4),(6).

The macro instructions which 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 in "Section III: List and Execute Forms" of the Supervisor and Data Management Macro Instructions manual. The expansion of the standard form of these macro instructions (that is, the form described in Section II of that publication) results in an in-line parameter list and executable instructions required 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 which 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 and main storage area 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 main storage assigned to the task through the use of the GETMAIN macro instruction. This ensures that the program remains reenterable.

Example 30 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; main 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 code 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,(4)) 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 4.

MOVERTN     GETMAIN  R,LV=(5),   Allocate main storage for list
                     HIARCHY=1   In IBM 2361 Core Storage
            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'
```

Example 30. Using the List and the Execute Forms of the DEQ Macro Instruction

NONREENTERABLE LOAD MODULES

The use of reenterable load modules does not automatically conserve main 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 main storage for the purpose of moving code from the load module to the allocated area is a waste of both time and main storage when only one task requires the use of the load module.

You may remember that, in an operating system with MVT, the area occupied by a reenterable or serially reusable load module is not released automatically when the module returns control to the control program. (Refer to "How Control is Returned" in the discussion of "Passing Control in a Dynamic Structure.") In anticipation of future use, the used copy of the module is retained intact for as long as possible; its area is available to fill both implicit and explicit requests for storage, but only after all other available storage has been allocated. If copies of several modules are retained when they are not needed, available storage may be fragmented as first the areas between the modules are allocated, and then the module areas themselves.

To prevent this fragmentation, 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. If reusability is not important, but you need to execute a module that has been made reusable, you can make the module temporarily nonreusable by bringing its directory entry into storage, modifying the contents of the entry, and using the entry to refer to the module. After issuing a BLDL macro instruction to build a list containing the directory entry, you need only set the first two bits of the twenty-third byte in the entry to zero; the module will then be treated as nonreusable when given control by a LINK, ATTACH, or XCTL macro instruction with a DE operand

that points to the entry. To set the appropriate bits to zero, you can use an AND-immediate instruction like the following, which could be placed after the BLDL macro instruction in Example 18:

```
NI NAMEADDR+22,B'00111111'
```

This instruction ensures the nonreusability of the module to which NAMEADDR refers.

One method of conserving main storage when reusability is not a consideration is to use a planned overlay structure. A complete description of the planned overlay structure is contained in the Linkage Editor and Loader manual. Briefly, in a planned overlay structure only portions of the load modules are brought into main storage at a time; when a portion of the load module not in main storage is required, it is loaded in the area occupied by existing portions of the load module. While the use of an overlay structure requires more planning on your part to determine all the portions of a load module required at any one time, it can result in a considerable saving of storage. A well planned overlay structure can result in a savings of 50 percent or more over bringing the entire load module into main storage at once. This does increase the amount of time spent in bringing in portions of the load module, however.

It is also possible for you to use an overlay type of approach in the design of your load module without using the linkage editor by reusing the areas containing completed routines within a load module. For example, if your load module consists of three control sections of 2000 bytes each which are always executed sequentially, as soon as control is passed to the second control section you have 2000 bytes (the size of the first control section) available to use as a data area. If you reuse this area, you can save up to 2000 bytes of additional main storage which would otherwise be allocated using DS instructions or GETMAIN macro instructions.

RELEASING MAIN STORAGE

As indicated in Program Management, the control program establishes two responsibility counts for every load module brought into main 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.

Except for those modules contained in the link pack area, the main storage area occupied by a load module is available for reuse 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 main storage use. Naturally, if you will use a load module many times in the course of a job step, you will issue a LOAD macro instruction to bring it into main storage, and you will not issue a DELETE macro instruction until all uses of the load module have completed. In this case it is better to have the load module in main storage all the time than to bring it in every time you require it.

Conversely, if a load module is used only once during the job step, or if its uses are widely separated, it will conserve main storage if you issue a LINK macro instruction to load the module and issue an XCTL from the module (or return control to the control program) when it has completed.

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 after the use is finished. If you do not issue a CLOSE macro instruction for the data control block, the control program will issue one for you when the task is terminated. However, if the load module containing the data control block has been removed from main storage, the attempt to issue the CLOSE macro instruction will cause 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 main storage when the task is terminated.

STORAGE HIERARCHIES

Main storage may be expanded by including IBM 2361 Core Storage in the system (excluding the Model 65 Multiprocessing System). Main Storage Hierarchy Support for IBM 2361 Models 1 and 2 permits selective access to either processor storage (storage associated with the Central Processing Unit) or IBM 2361 Core Storage. Processor Storage is referenced as hierarchy 0; IBM 2361 Core Storage is referenced as hierarchy 1. The first address in IBM 2361 Core Storage is one higher than the last address in processor storage.

Since IBM 2361 Core Storage is an extension of main storage, no special instructions are required for its use. Hierarchies 0 and 1 may be specified by using the hierarchy parameter (HIARCHY=) in the ATTACH, DCB, GETMAIN, GETPOOL, LINK, LOAD, and XCTL macro instructions. If the hierarchy parameter is omitted, requested storage, if available, is obtained from processor storage.

In using Main Storage Hierarchy support on a Model 50, use caution in directing programs containing CCWs for direct access devices to be loaded into hierarchy 1. (Under MFT, this includes readers and writers.) If this is disregarded, overrun will occur which will degrade the performance or result in an unrecoverable I/O error.

If IBM 2361 Core Storage is not included in an MFT system generated with storage hierarchies, requests for storage within hierarchy 1 are obtained from hierarchy 0. If IBM 2361 Core Storage is not included in an MVT system generated with storage hierarchies, the hierarchy structure is contained wholly within processor storage. Example 28 shows two GETMAIN requests for hierarchy 0. Example 29 shows a request for hierarchy 1. Requirements for writing macro instructions with the hierarchy parameter are described in the Supervisor and Data Management Macro Instructions manual.

When you submit a job for execution, you expect it to be executed quickly and efficiently. But if a job step terminates abnormally, you may have to submit the job again. You then lose valuable computer time and must wait longer for your results.

The operating system provides special facilities to reduce the effects of abnormal termination. When a job step terminates abnormally, you can restart it, either from the beginning or from a checkpoint within the job step itself. You can request that the restart automatically follow abnormal termination, or you can request restart later by submitting a new job.

When you submit a new job, you actually resubmit the original job with certain changes indicating where restart is to occur. If necessary, you can make more extensive changes, such as corrections to data that will be processed after restart. At times, you may wish to make such changes and then restart a job step that has terminated normally but has produced incorrect results.

When you restart a job step, the step may or may not be completed successfully. You can expect successful completion if abnormal termination was the result of a chance error, such as a parity error, because such an error should not recur after restart. If abnormal termination resulted from an error in data or job control statements, you can expect successful completion if you correct the error and request restart by submitting a new job. Obviously, you cannot expect successful completion if the cause of abnormal termination was an error in the logic of your program.

TYPES OF RESTART: You can request two basic types of restart:

- Step restart, which is a restart from the beginning of a job step.
- Checkpoint restart, which is a restart from a checkpoint within a job step. A job step can include any number of checkpoints. Each checkpoint is established by a CHKPT macro instruction.

You can request that either type of restart automatically follow abnormal termination. You can also request either type by submitting a new job.

AUTOMATIC RESTART: You request automatic step restart through job control statements; you request automatic checkpoint restart through the CHKPT macro instruction.

If you request automatic step restart, the job step will be restarted from the beginning if it terminates abnormally without issuing a CHKPT macro instruction. If the step terminates after issuing a CHKPT macro instruction, it will be restarted from the most recent checkpoint, unless automatic checkpoint restart is suppressed.

You can suppress automatic checkpoint restart through either a job control statement or the CHKPT macro instruction. If you do so, and you request automatic step restart, the job step will be restarted from the beginning in the event of abnormal termination. However, automatic step restart is also suppressed if abnormal termination occurs after restart from a checkpoint within the same step.

Automatic step or checkpoint restart is possible only when the abnormal completion code is one of a set of codes specified at system generation. (In a system with MFT or MVT, this set may include the code that represents a system failure requiring a system restart.) All automatic restarts must be authorized by the operator.

DEFERRED RESTART: Restart is deferred when you do not request automatic restart or when automatic restart is not allowed or is not successful. You request deferred restart by submitting a new job.

With deferred restart, you can consider the cause of abnormal termination, decide whether restart is likely to be successful, and make any necessary changes in data and job control statements. You can also decide whether to restart the job step from the beginning or from a checkpoint, and can choose a checkpoint other than the most recent one. In some cases, you may have the option of restarting the job step on an alternate computing system.

ESTABLISHING CHECKPOINTS

To establish a checkpoint, you use the CHKPT macro instruction. This macro instruction records the information necessary to restart the job step; it records this information in a checkpoint data set.

Checkpoint data sets are a special topic discussed later. The following discussion concerns the use of the CHKPT macro instruction, and the selection of checkpoints. You must be careful in selecting checkpoints, because their placement is important to successful restart.

In selecting a checkpoint, consider the following restrictions:

- When the checkpoint is established, the job step must comprise a single task. The job step task must be your only task when the job step is restarted.
- A checkpoint cannot be established by an exit routine that returns control to the control program. This type of routine is specified by the ATTACH, SPIE, and STIMER macro instructions, and by the EXLST and SYNAD operands of the DCB macro instruction. (There is one exception, a special EXLST routine that is discussed later.)
- If a STIMER or WTOR macro instruction has been issued, a checkpoint cannot be established before the time interval is completed or the operator's reply is received. After a restart, no timer interruption or operator reply could be expected.
- In a system with MVT and the rollout/rollin option, a checkpoint cannot be established when the job step has been allocated storage from outside its region.

In selecting a checkpoint, you must also consider the handling of data sets and serially reusable resources. First, however, it may help to consider how the CHKPT macro instruction is used to establish checkpoints.

Example 31 shows a CHKPT macro instruction and a DCB macro instruction for the checkpoint data set. The CHKPT macro instruction records information in the checkpoint data set and requests automatic restart if the job step later terminates abnormally. When the step is restarted, execution resumes with the instruction that follows the CHKPT macro instruction.

```
          ...
          CHKPT  CHKPTDCB
          ...
CHKPTDCB  DCB    DSORG=PS,MACRF=(W),RECFM=U,BLKSIZE=32760,     C
                 DDNAME=CHKPTDD
```

Example 31. Establishing a Checkpoint

When automatic restart is not possible, you can request a deferred restart by submitting a new job. The JOB statement for the new job refers to the checkpoint by an identification that (in Example 30) is generated by the control program and printed in a message to the operator.

After being restarted, the job step may again terminate abnormally. If it does, it may be automatically restarted from the same checkpoint, subject to operator authorization. To ensure that the job step is not restarted twice from the same checkpoint, you can code the sequence shown in Example 32.

The instruction that follows the checkpoint tests the return code register to determine whether control has been returned as the result of a restart. If the return code is four, a restart has just occurred, and a second CHKPT macro instruction is executed. This macro instruction has a CANCEL operand, which cancels the request of the previous macro instruction for an automatic restart. If the job step terminates abnormally after issuing CHKPT CANCEL, automatic restart can occur only at a later checkpoint. Because the step was restarted from a checkpoint, automatic restart cannot occur.

Restart from a checkpoint invalidates the results of certain macro instructions. One of these is the EXTRACT macro instruction which is used to obtain information from the task control block. This information is subject to change when the task is terminated and the job step is restarted. If the information is needed after restart, it should be updated by reissuing the EXTRACT macro instruction as shown in Example 33.

```
          ...
          CHKPT  CHKPTDCB  Establish checkpoint
          CH     15,=H'4'  Restart in progress?
          BNE    NRESTART  No, branch to NRESTART
          CHKPT  CANCEL    Yes, cancel restart request
NRESTART  ...
```

Example 32. Canceling a Request for Automatic Restart

```
          ...
          EXTRACT  ANSADDR,FIELDS=(ALL)  Obtain TCB information
          ...
          CHKPT    CHKPTDCB              Establish checkpoint
          CH       15,=H'4'              Restart in progress?
          BNE      NRESTART              No, branch to NRESTART
          EXTRACT  ANSADDR,FIELDS=(ALL)  Yes, obtain new information
NRESTART  ...
```

Example 33. Obtaining Updated TCB Information After Restart

Restart also invalidates the results of the ENQ and SETPRT macro instructions. The ENQ macro instruction, to be discussed in the next topic, is used to request control of serially reusable resources.

The SETPRT macro instruction is used in data management to load the Universal Character Set (UCS) buffer for a printer with the UCS feature and to load the forms control buffer (FCB) for a printer without a carriage control tape. The FCB and the carriage control tape both control paper movement in the printer. The contents of the buffers are not saved when a checkpoint is taken. To reload the buffers upon restart, you must reissue the SETPRT macro instruction in the same manner as the EXTRACT macro instruction.

CHECKPOINTS AND SERIALLY REUSABLE RESOURCES

When a job step terminates, it loses control of serially reusable resources. If the step is restarted, it must request all of the resources that it requires to continue processing.

Example 34 shows a program that requests a serially reusable resource before establishing a checkpoint. After the checkpoint, it conditionally requests the same resource. If the job step still has control of the resource, the control program ignores the request. It fills the request if the job step has terminated abnormally, has lost control of the resource, and has been restarted from the checkpoint.

SHARED DIRECT ACCESS STORAGE DEVICE

At some installations, a direct access storage device is shared by two or more independent computing systems. This device is a serially reusable resource; if it is being used when a checkpoint is taken, it must be requested after a restart from the checkpoint. This resource is requested not by the ENQ macro instruction, but by a special macro instruction (RESERVE) described in the System Programmer's Guide.

OTHER SERIALLY REUSABLE RESOURCES

There are some resources that you request implicitly by issuing data management macro instructions. These resources may be records that you are processing, or tracks on a direct access device. Since you cannot conditionally request control of these resources after a restart, you should not establish checkpoints while you have control of these resources.

- If you use the basic direct access method (BDAM), do not take a checkpoint before releasing a record that has been read with exclusive control. When you add a record to a data set, do not take a checkpoint before checking for completion of the write operation if the record format is variable-length or undefined.

```
...
ENQ    (QADDR,RADDR)
...
CHKPT CHKPTDCB
ENQ    (QADDR,RADDR),RET=HAVE
...
DEQ    (QADDR,RADDR)
...
```

Example 34. Requesting a Resource After Restart

- If you use the basic indexed sequential access method (BISAM), do not take a checkpoint before waiting for completion of a write operation. If you read a record for update, do not take a checkpoint before writing the updated record and waiting for completion of the write operation.

- If you use the queued indexed sequential access method (QISAM), issue an ESETL macro instruction before taking a checkpoint if you have previously issued a SETL macro instruction. You can issue another SETL macro instruction after the checkpoint.

CHECKPOINTS AND DATA MANAGEMENT

Data management is one of the most important considerations in selecting checkpoints, and is discussed in detail in the publication Data Management Services. The following discussion should be understandable if you have a basic knowledge of data management concepts and facilities.

DISPOSITION OF DATA SETS

At the end of a job step, data sets are disposed of according to your specifications in DD control statements. If a job step terminates abnormally, you should keep or catalog data sets that you may need for a deferred restart.

When you catalog a data set, you enable the operating system to retrieve the data set by name alone. You therefore do not have to provide volume and device-type information when you request deferred restart. Providing such information could require you to write new DD statements.

If you request automatic restart, the system keeps data sets for you, except when the restart is not actually performed. The kept data sets include "temporary" data sets and others that normally would be deleted. Data sets are deleted only if created by a job step that is to be restarted from the beginning.

Guidelines for specifying data set disposition appear in the topic "Using the Restart Facilities" in the Job Control Language Reference manual.

POSITIONING OF DATA SETS

If you take a checkpoint while processing a data set, you may continue processing for some time before abnormal termination. On restart, you must be able to resume processing at the correct location in the data set.

When the control program restarts a job step, it automatically repositions data sets on magnetic tape and direct access devices. It does not reposition data sets on unit record equipment; such data sets must be repositioned manually or by your program.

Unit Record Data Sets: Unit record output can be either punched cards or printed pages. Input can only be punched cards.

To reposition an output data set, you simply discard data punched or printed after a checkpoint. This data is recreated when the job step is restarted. Note that when pagination is important, you should take a checkpoint only after printing the last line on a page.

To reposition an input data set, you include a repositioning routine as part of your program. Such a routine should first determine whether repositioning is necessary, since the data set may have been transcribed onto a magnetic tape or direct access volume. If the data set has been transcribed, it is repositioned automatically by the control program; otherwise, it must be repositioned by your routine.

If you provide a repositioning routine, your program might operate as follows:

- The program saves the first record read from the data set and keeps a count of the total number of records read before each checkpoint.
- After a restart, the repositioning routine reads a record from the data set and compares it with the first record read before abnormal termination.
- If the records are identical, the data set has been positioned to the beginning. The routine repositions it by reading (without otherwise processing) the number of records read before the checkpoint.
- If the records differ, no repositioning is necessary. The data set presumably has been transcribed onto a magnetic tape or direct access volume, and has been repositioned by the control program.

Tape and Direct Access Data Sets: When the control program repositions a tape or direct access data set, it ensures that the correct volume is mounted. During an automatic restart, it may ask the operator to demount the current volume of a multivolume data set, and to replace it with an earlier volume. However, if the data set is physically sequential, you can ensure that it can be repositioned without changing volumes simply by taking a checkpoint each time a new volume is mounted. To do so, you provide a routine for taking a checkpoint, and specify its address in the data control block exit list. The control program gives control to this routine at the appropriate time. The requirements for writing an end-of-volume routine are described in "Processing Program Description," Section II, Part 1.

Positioning becomes especially important when you modify a physically sequential or partitioned data set (and specify DISP=MOD in the DD statement). In each case, you must take a checkpoint immediately after opening the data set, before writing any records. If you do not, errors will occur if:

- You take a checkpoint before opening the data set.
- You open the data set and begin writing records.
- The job step terminates and is restarted from the checkpoint.
- You reopen the data set after restart.

If you are using BISAM to add records to an ISAM data set, you must anticipate duplicate record indications following a restart. These duplicate record indications can occur when you attempt to add records that were already added before the restart. On the other hand, if you are using QISAM to add records to an ISAM data set, or if you are creating the data set, all records added after the checkpoint will be lost after the restart.

If you are modifying a sequential or partitioned data set, the data set will be positioned incorrectly when you reopen it after restart. Because of the parameter DISP=MOD, the data set is positioned to the end; that is, the data set is positioned after records that were added

prior to abnormal termination. Thus, records added after restart will duplicate those added before restart.

When you open a data set before taking a checkpoint, the data set is repositioned during a checkpoint restart. Also, when you specify DISP=MOD for a data set on a direct access device, the data set is repositioned (when opened) after an automatic step restart.

SYSIN and SYSOUT Data Sets: System input (SYSIN) data sets are data sets that you include with your job control statements in the system input stream. System output (SYSOUT) data sets are data sets that you route to a printer or card punch through the system output stream. By routing data sets through the input and output streams, you avoid having to request unit record devices for exclusive use by your job step.

A SYSIN or SYSOUT data set may or may not be on a unit record device at the time it is processed by your program. In a system with MFT or MVT, a SYSIN data set is always on a direct access device, while a SYSOUT data set may be on a unit record device, magnetic tape unit, or direct access device. Transcription from one type of device to another (such as card-to-tape transcription for SYSIN data sets) is handled by the operator or the operating system.

When a job step is restarted, the repositioning of a SYSIN or SYSOUT data set depends on the type of device that is actually used by your program. If the device is a unit record device, you must reposition the data set yourself just as you do any other unit record data set. If the device is a magnetic tape unit or direct access device, the data set is repositioned automatically.

A SYSOUT data set has the implied status DISP=MOD. Therefore, a checkpoint should be taken immediately after a SYSOUT data set is opened. For automatic step restart, the implied status DISP=MOD means that SYSOUT data sets on magnetic tape are not repositioned in the same way as SYSOUT data sets on direct access devices. SYSOUT data sets on tape are positioned to the end; SYSOUT data sets on direct access devices are positioned to the beginning.

For deferred checkpoint restart, note that:

- If a SYSIN data set was read completely before the checkpoint, you need not include the data set when you request restart from the checkpoint. If only part of the data set was read, you must include the complete data set so that it can be properly repositioned.
- If the checkpoint was taken while a SYSIN or SYSOUT data set was being processed, the type of device used directly by your program must be the same for restart as for original execution. The blocking factor (number of records per block) must also be the same.

PRESERVATION OF DATA SETS

The control program repositions data sets but does not preserve their contents. After taking a checkpoint, you must ensure that the data set contents are not changed in a manner that would make successful restart impossible.

If you read records from a data set, update them, and write them back to their original locations, it may be useless to take a checkpoint before completing this processing. If you take a checkpoint earlier, restart will produce invalid results if you update a record before abnormal termination, update it again after restart, and actually change the record in both cases. For example, suppose the purpose of the update is to switch the positions of two fields in each record. If you

update a record twice, you return the fields to their original positions, and the results are invalid. In a different application, an update might simply place a value in a record field, regardless of the field's original contents. In this case, you could restart the step at a checkpoint taken before or during the update procedure, because an updated record would not be changed if updated again after restart.

Partitioned Data Sets: When you process a partitioned data set, you must be careful to preserve the contents of the directory. The directory consists of entries that point to each member of the data set.

When you add a member to a partitioned set, you also add an entry to the directory. If you add only one member, you can use the STOW macro instruction to create the entry, or you can specify the member name in the DD statement; in the latter case, the control program creates the directory entry when you close the data set or when the job step terminates. If you add more than one member, you must use the STOW macro instruction to create an entry for each member.

When you add one or more members to a partitioned data set, you must take a checkpoint immediately after opening the data set. After taking the checkpoint, you can write the new member and add its entry to the directory. Then, if the step is restarted from the checkpoint, the data set is repositioned; the new member and its directory entry are deleted, and are recreated after restart.

If you do not take a checkpoint after opening the data set, various errors may occur. As an example, assume that:

- You take a checkpoint before opening a partitioned data set.
- You open the data set and begin writing a new member.
- The step terminates abnormally; the control program creates a directory entry for the new member, using the member name specified in the DD statement.
- The step is automatically restarted from the checkpoint; the data set is not open, and therefore it is not repositioned.
- You reopen the data set after restart; the control program positions the data set after the member that was just created.
- You write the member again and close the data set; the control program tries to create a directory entry, again using the member name specified in the DD statement.

The attempt to create a directory entry after restart is unsuccessful, because the member name already appears in the entry that was created before abnormal termination. The step again terminates abnormally, and the member created after restart is deleted.

Note that when a partitioned data set is repositioned after restart from a checkpoint, the control program deletes all members that have been added to the data set since the checkpoint was taken. You therefore should not request a deferred checkpoint restart if it would delete members that have been added by other jobs.

To update a member of a partitioned data set, you can either write updated records back to their original locations, or rewrite the entire member (in updated form) as a new member of the data set. In the latter case, you update the directory entry to point to the rewritten member.

If you take a checkpoint before rewriting a member, you must also take one immediately after updating the directory. You must do so

because the control program will delete the updated directory entry if it repositions the data set for restart from the earlier checkpoint. Since no entry then points to the original member, execution after restart will be unsuccessful.

Data Sets on Direct Access Devices: For every data set on a direct access device, there is a standard data set label called a data set control block (DSCB). The DSCB is part of the volume table of contents (VTOC); it defines the location and extent of the data set on a particular volume.

If you take a checkpoint while processing a data set on a direct access device, the job step can be restarted from the checkpoint only if the DSCB has not been changed since the checkpoint was taken, or if the only changes result from:

- Secondary allocation. In the DD statement, you can request that additional space be allocated to the data set when the space currently available is exhausted. If space is allocated after a checkpoint is taken, this space is indicated in the DSCB; on restart from the checkpoint, the space is released and the DSCB is changed accordingly.

- Release of unused space. In the DD statement, you can request that unused space be released at the end of the job step. If space is released, the DSCB may indicate a reduced extent for the data set when checkpoint restart is deferred; no space is allocated to replace that which was released. Note that space is not released when step termination is followed by automatic restart.

If the DSCB is changed by moving the data set to a new location on the same volume, or by moving the data set to a new volume, the job step cannot be restarted from the checkpoint unless:

- Restart is deferred.

- The data set is replaced by a dummy data set. (Refer to the discussion of "Dummy Data Sets" below.)

If a data set occupies more than one volume, there is a DSCB for the data set on each volume. If the data set is processed sequentially, only one volume is being processed when the checkpoint is taken; if the DSCB for this volume has not been changed, the job step can be restarted from the checkpoint even though there may be changes in the DSCBs for the data set on other volumes.

When end-of-volume is reached in writing a data set, secondary allocation may cause the data set to be continued on another volume. If the allocation occurs after a checkpoint, the volume used for continuation will not be mounted on restart from the checkpoint. The control program therefore cannot release the allocated space, even though it no longer recognizes this space as a part of the data set.

To release space on a volume that is not mounted on restart, you should use a utility program to delete the extension of the data set on the volume. If you do not release the space before the job step is restarted, the step will be abnormally terminated if the data set is again extended to the same volume. Note that if the data set organization is physically sequential, you can provide an end-of-volume exit routine to ensure that a checkpoint is taken each time the data set is extended to a new volume.

Work Data Sets: Many programs use "work" data sets, which are alternately written and read, rewritten and reread. If you use a work data set, you should take a checkpoint each time you have finished

reading the data set, before rewriting it. Then, if the job step is restarted, you will not need to read records that you have destroyed by rewriting the data set. If you use the data set many times, you can reduce the frequency of checkpoints by using two data sets, as shown in Example 35. If you use two data sets on separate volumes, you can assign both to one device through the UNIT parameter in the associated DD control statements.

Dummy Data Sets: When you request deferred checkpoint restart, you can sometimes use dummy data sets to replace data sets that were used during the original execution of your program. For example, your program may have taken a checkpoint while processing a data set; it may have finished processing the data set prior to abnormal termination, or the data set may have been deleted. If there is no need to process the data set after restart, you can replace it with a dummy data set, provided that:

- The data set is sequentially organized and is processed by the basic or the queued sequential access method (BSAM or QSAM).
- The job step is not restarted from a checkpoint that is within the data set's end-of-volume exit routine.

Of course, the data set must not be the checkpoint data set that is being used to restart the job step.

After restart, an input request for a dummy data set results in an immediate end-of-data-set condition. An output request is processed normally, except that no data is actually written.

You define a dummy data set by means of a DD statement containing the parameter DUMMY or DSNAME=NULLFILE. The name of the DD statement must be the same as that of the DD statement for the data set being replaced.

PRE-ALLOCATED DATA SETS: In systems with MVT, direct access space for temporary data sets can be pre-allocated to save time. However, you cannot use this facility with checkpoint/restart. Checkpoints and automatic restarts are suppressed for any job step that uses a pre-allocated temporary data set.

Pre-allocated data sets are discussed in detail in the chapter "System Reader, Initiator and Writer Cataloged Procedures" in the publication System Programmer's Guide.

Using One Data Set (A)	Using Two Data Sets (A1 and A2)
Open A	Open A1
Write and read back A	Write and read back A1
Checkpoint	Close A1 and open A2
Rewrite and read back A	Write and read back A2
Checkpoint	Checkpoint
Rewrite and read back A	Rewrite and read back A2
Checkpoint	Close A2 and open A1
Rewrite and read back A	Rewrite and read back A1
Close A	Close A1

Example 35. Checkpoints for Processing Work Data Sets

CHECKPOINT DATA SETS

When you establish a checkpoint, the control program creates an entry in a checkpoint data set. The entry contains the information necessary to restart the job step from the checkpoint.

DEFINING A CHECKPOINT DATA SET

To define a checkpoint data set, you use the DCB macro instruction. This macro instruction creates a data control block, which describes the data set to the control program. The data control block contains information that you specify in the DCB macro instruction or in a DD job control statement.

The DCB macro instruction must specify the data set organization and the type of instruction that the control program will use to write entries in the data set. Other information, such as block size and record format, can be specified either in the DCB macro instruction or in the DD statement. Some information is optional and some required; the following examples provide all of the required information that can be coded in the macro instruction:

```
D1 DCB DSORG=PS,MACRF=(W),RECFM=U,BLKSIZE=32760,DDNAME=CHECKDD1

D2 DCB DSORG=PO,MACRF=(W),RECFM=U,BLKSIZE=600,DDNAME=CHECKDD2
```

A checkpoint data set must be physically sequential (DSORG=PS) or partitioned (DSORG=PO), and must be processed using the WRITE macro instruction (MACRF=(W)). The record format must be undefined (RECFM=U). The block size must be at least 600 bytes (BLKSIZE=600), but not greater than 32,760 bytes for magnetic tape, and not greater than the track length for direct access. You can omit block size information if you allow the control program to open the data set (as discussed in the next topic); in this case, the control program determines the maximum block size for the device being used, and places it in the data control block.

The data control block must refer to a DD statement (DDNAME=CHECKDD1, for example) for such additional information as the data set name and the type of labels used for magnetic tape. (A tape can have standard labels, nonstandard labels, or no labels.)

For seven-track tape, you must specify the tape recording technique (TRTCH=C, data conversion with odd parity). If you specify it in the DCB macro instruction, you must also specify device dependency (DEVD=TA). For direct access, you must not specify key length unless you specify a length of zero (KEYLEN=0).

As an optional service, you can request chained scheduling of input/output operations (OPTCD=C and NCP=2 channel programs). With direct access, you can request validity checking for write operations, with or without chained scheduling (OPTCD=WC or OPTCD=W). With direct access and normal scheduling, you can request use of track overflow (RECFM=UT).

USING A CHECKPOINT DATA SET

Before any data set can be used, it must be opened by issuing the OPEN macro instruction. When you use a checkpoint data set, you can open it yourself or allow the control program to open it for you. If the data set is not open when you issue the CHKPT macro instruction, the control program opens it, writes a checkpoint entry, and then closes the data set before returning control to your program.

If you open the checkpoint data set yourself, you need not close it until after taking the last checkpoint for the job step. If you take many checkpoints, you will save a considerable amount of time if you allow the data set to remain open. You will also save all of the checkpoint entries and thus be able to request a deferred restart from any of the checkpoints.

If the control program opens the data set, the data set is positioned for each checkpoint according to your specifications in the DD statement. If you specify DISP=MOD, the data set is positioned to the end and each entry is written after that for the previous checkpoint. If you specify anything else, the data set is positioned to the beginning and each entry is written over the previous entry.

By allowing the control program to write over a previous entry, you can save space in external storage. You should not allow it to write over the most recent entry, however, because the job step might be terminated while the new entry was being written. To save the most recent entry, you can use two checkpoint data sets in alternation; the new entry is then written in one data set while the previous entry is saved in the other.

Example 36 shows a way of alternating data sets when all checkpoints are taken by one CHKPT macro instruction. The data sets are opened by the control program, and are identified by two DD statements, CHECKDD1 and CHECKDD2. The data control block initially refers to CHECKDD2, but is changed before the first checkpoint to refer to CHECKDD1. Before the second checkpoint, it is changed to refer to CHECKDD2; before the third checkpoint, it is again changed to refer to CHECKDD1, and so forth. In this way, one data control block can be used for two data sets that are not open at the same time. (The DCBD macro instruction, used in Example 36, is described in the section "Modifying the Data Control Block," of Data Management Services.)

With direct access, a checkpoint data set must be written entirely on one volume. Also, it must be written entirely in the space originally allocated to the data set. When the available space cannot contain a complete checkpoint entry, an attempt to take a checkpoint results in abnormal termination, unless you have requested secondary space allocation in the DD statement. If you have requested secondary allocation, abnormal termination does not occur, even though the space cannot be used. Control is returned to your program with an error indication in register 15.

With magnetic tape, a checkpoint data set can be written on more than one volume. If end-of-volume is reached in writing an entry, the entire

```
            ...
            DCBD   DSORG=PS             Define IHADCB (dummy section
            CSECT                         that defines DCBDDNAM)
            ...
            LA     2,CHECKDCB           Establish CHECKDCB as base
            USING  IHADCB,2               address for IHADCB
            XC     DCBDDNAM(8),DDHOLD   Exchange ddname in CHECKDCB
            XC     DDHOLD(8),DCBDDNAM     for ddname in DDHOLD
            XC     DCBDDNAM(8),DDHOLD
            CHKPT  CHECKDCB             Open, checkpoint, close
            ...
DDHOLD      DC     C'CHECKDD1'
CHECKDCB    DCB    DSORG=PS,MACRF=(W),DDNAME=CHECKDD2
```

Example 36. Alternating Use of Checkpoint Data Sets

entry is written on the next volume. The volume that contains the complete entry is indicated in the message that identifies the checkpoint.

Note that you must use a checkpoint data set only for taking checkpoints. If you use a data set for any other purpose, you cannot use it as a checkpoint data set.

RESTARTING A JOB STEP

If you request an automatic restart, the control program uses the most recent entry in the checkpoint data set (or the most recent valid entry if an uncorrectable error occurred in writing the most recent entry). If you request a deferred restart, you must specify the appropriate checkpoint entry when you submit the job for restart.

DEFERRED RESTART

To identify the checkpoint data set, you include an appropriate DD statement after the JOB statement, or after the //JOBLIB DD statement if you define a job library. The name of the statement must be SYSCHK.

In the JOB statement, you specify the name of the job step to be restarted and the checkpoint at which restart is to occur. You specify the checkpoint by an identification that was printed on the operator's console when the checkpoint was taken.

CHECKPOINT IDENTIFICATION

The control program assigns the identification for each checkpoint, unless you assign it yourself when you issue the CHKPT macro instruction. Example 37 shows a macro instruction that assigns the identification "ENDOFDATAONINPUT". The identification is 16 characters in length -- the maximum length allowed for a physically sequential data set. For a partitioned data set, the identification is used as a member name and, therefore, cannot exceed eight characters.

If you assign checkpoint identifications, you should not assign the same identification to two or more checkpoints. If you do, you will be able to restart the job step from only one of the checkpoints if you save the entries in the same checkpoint data set. In the case of a physically sequential data set, you can restart the step only from the earliest checkpoint, because the control program will find its entry first when it searches the data set. In the case of a partitioned data set, you can restart the step only from the latest checkpoint, because its entry is a member of the data set and replaces any previous entry with the same identification (member name).

When the control program assigns identifications, the identification for each checkpoint is unique. The identification is eight bytes in length, and consists of the letter C followed by a seven-digit decimal

```
          ...
          CHKPT  CHECKDCB,CHECKID3,16
          ...
CHECKID3  DC     C'ENDOFDATAONINPUT'
CHECKDCB  DCB    DSORG=PS,MACRF=(W),DDNAME=CHKDD
```

Example 37. Assigning a Checkpoint Identification

```
          ...
          CHKPT  CHKDCB,ID,'S'     Take checkpoint
          LTR    15,15             Checkpoint taken?
          BNZ    PHASE2            No, branch to PHASE2
          PUT    STEPLOG,MESSAGE   Yes, print checkpoint ID
PHASE2    ...
          ...
MESSAGE   DC     H'45,0'           Record length, etc.
          DC     C'SUCCESSFUL CHKPT AT PHASE2.  ID='
ID        DS     CL8
STEPLOG   DCB    DSORG=PS,MACRF=(PM),RECFM=V,BLKSIZE=128,      C
                 LRECL=124,DDNAME=LOGDD
CHKDCB    DCB    DSORG=PS,MACRF=(W),RECFM=1,BLKSIZE=32760,     C
                 DDNAME=CHKDD
```

Example 38. Recording a Checkpoint Identification Assigned by the Control Program

number. The number is the total number of checkpoints taken by the job, including the current checkpoint, checkpoints taken earlier in the job step, and checkpoints taken by any previous job steps.

The control program identifies each checkpoint in a message to the operator; on request, it also makes the identification available to your program. In Example 38, the CHKPT macro instruction requests the control program to supply an identification and place it in the eight-byte field named ID. When the checkpoint is successfully taken, the program prints the identification as part of a message to the programmer.

RESTART ON AN ALTERNATE SYSTEM

You can request deferred restart on a system other than the one on which your job was originally executed. Of course, the alternate system must have facilities adequate to process your job, and, in the case of checkpoint restart, it must be identical in certain respects to the original system.

- The type of operating system (MFT or MVT) must be the same for both systems. Also, the release level must be the same.
- The nucleus of the alternate system must be identical to that of the original system.
- The main storage area available to your job step must be the same in both systems.
- If your job step uses data management access methods, the resident routines for these access methods must have the same main storage locations in both systems. In systems with MVT, these routines are located in the link pack area. If your job step uses other modules in the link pack area, these modules must also have the same locations in both systems.
- If your job step uses main storage hierarchy 1, the boundary between hierarchies 0 and 1 must be the same in both systems.

FURTHER INFORMATION ON RESTART

For further information on restart, refer to the topic, "Using the Restart Facilities" in the Job Control Language Reference manual.

The operating system with the time sharing option (TSO) provides certain services in addition to the ones discussed above.

SPECIFYING AN ATTENTION EXIT ROUTINE

Use the STAX macro instruction to specify the address of an attention exit routine to gain control when the terminal user strikes the attention key or when the terminal user specifies simulated attention. The details about what you should do in an attention exit routine and how you can use it appear in the Time Sharing Option Guide to Writing a Terminal Monitoring Program or a Command Processor.

MANIPULATING TASK PROCESSING

Use the STATUS macro instruction to specify that a task is or is not to be dispatched by the system.

When you issue the STATUS macro instruction with the START or STOP operand, the system determines whether the specified subtask of the current task or all subtasks of the current task are to be modified. When you specify START, the stop/start count in the subtask TCB(s) is decreased and the nondispatchability flags are cleared. When you specify STOP, the stop/start count in the subtask TCB(s) is increased and the nondispatchability flags are set. The nondispatchability flags are set for a task only if the task has no system routine being executed for it. If a system routine is being executed for the task, the task is made nondispatchable when it no longer has a system routine being executed for it.

INDEX

Indexes to Systems Reference Library publications are consolidated in IBM System/360 Operating System: Systems Reference Library Master Index, GC28-6644. For additional information about any subject listed below, refer to other publications listed for the same subject in the Master Index.

IBM

International Business Machines Corporation
Data Processing Division
112 East Post Road, White Plains, N.Y. 10601
[USA Only]

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
[International]

READER'S COMMENT FORM

IBM System/360 Operating System
Supervisor Services

GC28-6646-5

Please check or fill in the items below, adding explanations and other comments in the space provided.

Which of the following terms best describes your job?

☐ Programmer	☐ Systems Analyst	☐ Customer Engineer
☐ Manager	☐ Engineer	☐ Systems Engineer
☐ Operator	☐ Mathematician	☐ Sales Representative
☐ Instructor	☐ Student/Trainee	☐ Other (explain) _____

Does your installation subscribe to the SRL Revision Service? ☐ Yes ☐ No

How did you use this publication?

- ☐ As an introduction
- ☐ As a reference manual
- ☐ As a text (student)
- ☐ As a text (instructor)
- ☐ For another purpose (explain) _____

Did you find the material easy to read and understand? ☐ Yes ☐ No (explain below)

Did you find the material organized for convenient use? ☐ Yes ☐ No (explain below)

Specific criticisms (explain below)

Clarifications on pages _____

Additions on pages _____

Deletions on pages _____

Errors on pages _____

Explanations and other comments:

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

YOUR COMMENTS PLEASE . . .

This manual is one of a series which serves as reference sources for systems analysts, programmers and operators of IBM systems. Your answers to the questions on the back of this form, together with your comments, will help us produce better publications for your use. Each reply will be carefully reviewed by the persons responsible for writing and publishing this material. All comments and suggestions become the property of IBM.

Please note: Requests for copies of publications and for assistance in utilizing your IBM system should be directed to your IBM representative or to the IBM sales office serving your locality.

FOLD FOLD

FIRST CLASS
PERMIT NO. 116
KINGSTON, N. Y.

BUSINESS REPLY MAIL
NO POSTAGE STAMP NECESSARY IF MAILED IN U. S. A.

POSTAGE WILL BE PAID BY

IBM CORPORATION
NEIGHBORHOOD ROAD
KINGSTON, N. Y. 12401

ATTN: PROGRAMMING PUBLICATIONS
DEPARTMENT 636

FOLD FOLD

IBM®

International Business Machines Corporation
Data Processing Division
112 East Post Road, White Plains, N.Y. 10601
[USA Only]

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
[International]

OS supervisor services Printed in U.S.A. GC28-6646-5