File No. S360-36
Order Number GC28-6646-3

OS

19

IBM Systems Reference Library

IBM System/360 Operating System
Supervisor and Data Management Services

Written for the assembler programmer, this publication describes the services and facilities available in the IBM System/360 Operating System to the user of the supervisor and the data management macro instructions. It also describes the linkage conventions that the user should use with the operating system. Macro instructions used for graphics, teleprocessing, optical readers, optical reader-sorters, or magnetic character readers are included in separate publications. These publications are listed in IBM System/360 Bibliography, Form A22-6822.

This publication covers the three main configurations of the operating system: systems with the primary control program (PCP); systems that provide multiprogramming with a fixed number of tasks (MFT); and systems that provide multiprogramming with a variable number of tasks (MVT).

Fourth Edition (June, 1970)

This publication corresponds to Release 19. It is a major revision of Form C28-6646-2, which is now obsolete. Most of the changes to the text and some of the changes to the illustrations are indicated by a vertical line in the margin to the left of the change. However, a new or extensively changed illustration is indicated by the symbol • to the left of the illustration's caption. Similarly, extensive changes to the text are indicated by the symbol • beside the page number.

Many changes to the book are listed in the section "Summary of Changes." However, this summary does not include all changes; there are technical changes throughout the book.

Specifications contained herein are subject to change from time to time. Before using this manual with IBM systems, consult the latest IBM 360 SRL Newsletter, Form N20-0360, for the editions that are current and applicable.

This publication was prepared for production using an IBM computer to update the text and to control the page and line format. Page impressions for photo-offset printing were obtained from an IBM 1403 Printer using a special print chain.

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 D78, San Jose, California, 95114. All comments become the property of IBM.

Preface

This publication describes the supervisor services and data management facilities of the IBM System/360 Operating System. It is written for the assembler programmer who is designing a program using these services and facilities. When coding an assembler program, however, the programmer needs the specific information in the publication IBM System/360 Operating System: Supervisor and Data Management Macro Instructions.

The publication is divided into three principal parts. Each section has a format designed to fit the illustrations and examples required to explain the subject.

- Supervisor Services -- This section covers the supervisor services available through the use of assembler language macro instructions, and describes linkage conventions, requirements for program and main storage management, the program management services available, and task creation and management.
- Data Management Services -- This section covers the data management services available through the use of assembler language macro instructions, and describes the data organization and access features of the operating system, along with cataloging and space allocation facilities.
- Appendixes -- Information is presented on the format and use of direct data set labels and on control characters.

PREREQUISITE PUBLICATIONS

IBM System/360 Operating System:

Assembler Language, Form C28-6514

Concepts and Facilities, Form C28-6535

PUBLICATIONS TO WHICH THE TEXT REFERS

IBM System/360 Operating System:

Job Control Language Reference, Form C28-6704

Job Control Language User's Guide, Form C28-6703

Linkage Editor and Loader, Form C28-6538

Model 91 Functional Characteristics, Form A22-6907

Model 195 Functional Characteristics, Form A22-6943

Programmer's Guide to Debugging, Form C28-6670

Storage Estimates, Form C28-6551

Supervisor and Data Management Macro Instructions, Form C28-6647

System Control Blocks, Form C28-6628

System Programmer's Guide, Form C28-6550

System Generation, Form C28-6554

Tape Labels, Form C28-6680

Summary of Changes

Following is a list of the programming changes that affect the elease 19 version of the manual IBM System/360 Operating System: upervisor and Data Management Services.

Reasons for Changes:	Items Changed or Added:
American National Standard COBOL	"Spanned Variable-Length Records (Sequential Access Methods)" "Spanned Variable-Length Records (Basic Direct Access Method)" "Block Size (BLKSIZE)" Table 9 "Standard User Label Exit" "Defer Nonstandard Input Trailer Label Exit" "PUTX -- Write an Updated Record" "READ -- Read a Block" "Data Event Control Block (DECB)" "CLOSE -- Terminate Processing of a Data Set" "FEOV -- Force End of Volume" "BUILDRCD -- Build a Buffer Pool and a Record Area" "Buffer Control" Table 13
The IBM System/360 Model 195	"Precise and Imprecise Interruptions" Table 6 Table 7

(Continued)

Reasons for Changes:	Items Changed or Added:
The IBM 1285, 1287, and 1288 Optical Character Readers, the IBM 1275 Optical Reader Sorter, and the IBM 1419 Magnetic Character Reader	Abstract (on front cover)
The ATLAS macro instruction	"ATLAS -- Perform Alternate Track Location Assignment
The ability to reuse previously allocated space and the full track index write option, both in ISAM	"Creating an Indexed Sequential Data Set
Different density defaults for magnetic tape	"Magnetic Tape (TA)" Table 14
Writing data sets directly onto the SYSOUT device under MFT and MVT	"SYSIN Data Sets" "Routing Data Sets Through the Output Stream"
The ATTACH macro instruction under MFT without subtasking	"Section I: Supervisor Services"
Using the WTO and the WTOR macro instructions to write messages to the programmer	"Writing to the Programmer"
Time slicing in MVT with the IBM System/360 Model 65 Multiprocessing System	"Time Slicing"

Following are some other changed topics:

- "Main Storage Control"
- "Data Sets on Direct Access Devices"
- "Dummy Data Sets"
- "Indexed Sequential Data Set Organization"
- "Writing a SYSOUT Data Set"

Contents

Illustrations

Figures

Tables

Examples

Section I: Supervisor Services

Introduction

The supervisor services section of this publication describes the supervisor services available from the IBM System/360 Operating System through the use of the supervisor macro instructions supplied by IBM. The information in this section includes a discussion of the standard linkage conventions to be used with the operating system, as well as a discussion of the requirements for using the macro instructions. This publication is to be used when designing a program; the information required to code the macro instructions is presented in the companion publication IBM System/360 Operating System: Supervisor and Data Management Macro Instructions.

This section covers the three major configurations of the operating system: the operating system with the primary control program (PCP); 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 three configurations of the operating system is given in Table 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.

Program Management

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

Table 1. Summary of Characteristics and Available Options

	PCP	MFT	MVT
Brief Description	Sequential Scheduler, one task per job step, one job processed at a time	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	Optional	Standard	Standard
Identify Option	Optional	Optional	Standard
Time Option	Optional	Optional	Standard
Interval Timing Option	Optional	Optional	Standard
System Log Option	Not available	Optional	Optional

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 1. When 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 1, 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 insures that later changes to your program will not result in the modification of the contents of a register which has not been saved.

Word	Contents
1	Used by PL/1 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 1. Save Area Format

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.

```
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

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 your program is going to use any system macro instructions (other than SAVE, RETURN, or the register forms of GETMAIN and FREEMAIN), or 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 your program does not use system macro instructions and 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.

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.

```
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

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 2, which is declared as a base register. The contents of register 1 are saved in another register, and a GETMAIN 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.

```
PROGNAME   SAVE    (14,12)
           LR      2,15
           USING   PROGNAME,2
           LR      3,1
           GETMAIN R,LV=72
           ST      13,4(1)
           ST      1,8(13)
           LR      13,1
           ...
```

Example 6. Reenterable Save Area Chaining

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

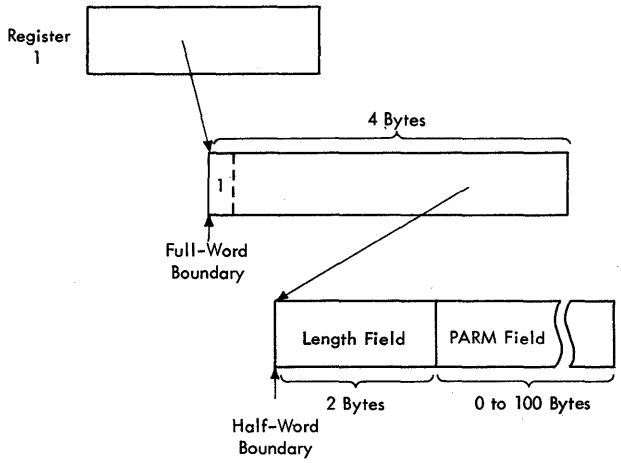


Figure 2. Acquiring PARM Field Information

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 does not pass control to any 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.

Table 2 summarizes the characteristics of these load module structures.

Table 2. Load Module Characteristics

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

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 PCP; there is only one task in the job step. Execution is also serial in an operating system with MFT and 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 publication IBM System/360 Operating System: Assembler Language 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.

```
          ...
          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

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

```
          ...
          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
          ER     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

register, restoring the registers earlier would have made the parameter list unaddressable.) A branch instruction using register 15 passes control to entry point NEXT.

PASSING CONTROL WITH RETURN

The control program passed control to your program, and your program will return control when it is through processing. Similarly, control sections within your program will pass control to other control sections, and expect to receive control back. An example of this type of control section is a "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

```
          ...
          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

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 tnis example is twelve full words; the size of the area for any specific application depends on the requirements of the two control sections involved.

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

Example 10. Passing Control With CALL

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.

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 publication IBM System/360 Operating System: Linkage Editor and Loader, 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 publication IBM System/360 Operating System: Supervisor and Data Management Macro Instructions.

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.

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

```
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

```
          ...
          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

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 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. Whether you were using an operating system with PCP, MFT, or MVT has not affected the previous discussion. 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 publication IBM System/360 Operating System: Linkage Editor and Loader.

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, 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 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.

- 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.

If you are using an operating system with MFT or MVT, some of the load modules from the link library may already be in main storage in an area called the resident reenterable module area (MFT) or the link pack area (MVT). However, the resident reenterable module area is optional in an operating system with MFT. The contents of this area 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. In an operating system with MFT, the resident reenterable module area can contain user-written modules and the loader, discussed in the publication IBM System/360 Operating System: Linkage Editor and Loader.

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. 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 libraries, the job pack area, and, when one exists, the link pack area. 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 and the job or step 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. Table 3 shows the order of the search when EP or EPLOC is coded, and the DCB operand is omitted or DCB=0 is coded.

Table 3. Search for Module, EP or EPLOC Operands With DCB=0 or DCB Operand Omitted

PCP	MFT	MVT
The job pack area is searched for an available copy	The partition is searched	The job pack area of the region is searched for an available copy
The step library is searched; if there is no step library, the job library (if any) is 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 link library is searched	The job library (if any) is searched	The link pack area is searched
	The link library is searched	The link library is searched

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 Table 4.

When used without the DCB operand, the EP and EPLOC operands provide the easiest method of requesting a load module from the link, job, or step library. The job or step library is searched before the link library, and the data sets that make up this library are searched in the order of their DD statements. Thus, one library or data set within a library 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 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,---
```

Searching a job or step library slows the retrieval of load modules from the link library; to speed this retrieval, you should limit the size of the job and step

Table 4. Search for Module, EP or EPLOC Operands With DCB Operand Specifying Private Library

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

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 assembly) 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.

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 Table 5 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 the primary control program or 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 Table 3, Table 4, and Table 5. Examples of the use of these tables are shown in the discussion 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.

Table 5. Search for Module Using DE Operand

PCP	MFT	MVT
Directory Entry Indicates Link Library and DCB=0 or DCB Operand Omitted		
The job pack area is searched for an available copy	The partition is searched	The job pack area for the region is searched for an available copy
The module is obtained from the link library	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 Library and DCB=0 or DCB Operand Omitted		
The job pack area is searched for an available copy	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	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 is searched for an available copy	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	The module is obtained from the specified private library

Operating System With MVT: If you are using an operating system with MVT, all reenterable modules (modules designated as reenterable using the linkage editor) from any library are completely reusable; only one copy is ever placed in the link pack area or brought into your job pack area, and you get immediate control of the load module. If the module is serially reusable, only one copy is ever placed in the job pack area; this copy 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 System 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.

PCP and MFT Systems Without Subtasking: If you are using an operating system with PCP or 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.

USE OF 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 without subtasking (that is, PCP or 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 Table 3.

```
          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    X'0001'        Number of list entries
          DC    X'003A'        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

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 (hexadecimal 3A). 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.

USE OF THE ATTACH MACRO INSTRUCTION (PCP AND MFT WITHOUT SUBTASKING): This discussion applies only if you are using an operating system with the primary control program or 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 PCP or 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 the primary control program or 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 (PCP or 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.

USE OF 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. 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 unpredicatable results if you try to return control to it.

Figure 3 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 3 indicates the result.

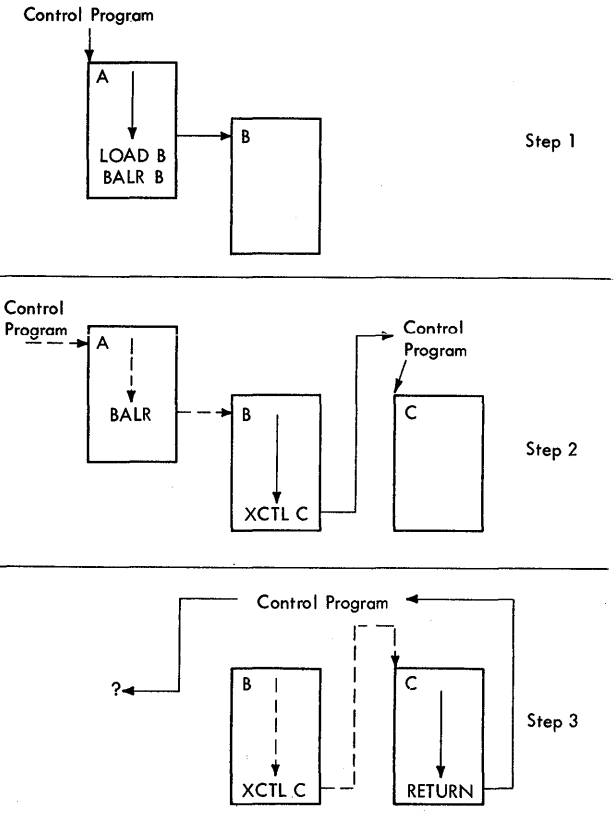


Figure 3. Misusing Control Program Facilities

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 PCP or 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 which 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.

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, and the operands that provide for communication between tasks.

Note: Although you are using an MFT system, you can include the subpool and rollout/rollin parameters for compatibility with an MVT system. If you code these parameters correctly, a system with MFT ignores them.

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:

Dispatching Priority =
(value$_1$ x 16) + value$_2$

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

Dispatching Priority =
(6 x 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. The use of the ATTACH and the CHAP macro instructions can affect dispatching priorities, as in MVT systems.

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.

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.

Task Management

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 4.

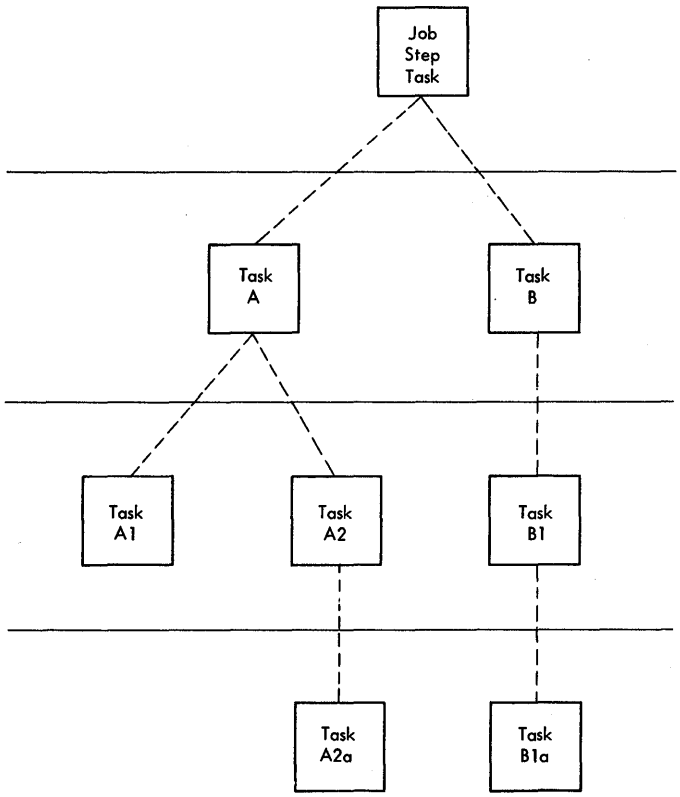


Figure 4. Task Hierarchy

The horizontal lines in Figure 4 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.

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 that 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 5.

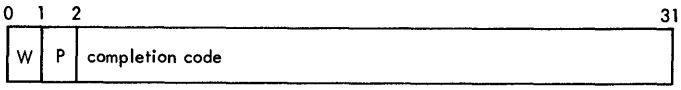


Figure 5. Event Control Block

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 POST or WAIT macro instruction can be issued. 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.

Program Management Services

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 either MFT or the primary control program: 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 the primary control program or 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 publication IBM System/360

Operating System: 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 be used to test for simultaneous use within one task in an operating system with MFT or MVT only. The ENQ and DEQ macro instructions are not available in an operating system with the primary control program.

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 6 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 6, 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 6, 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 6, 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 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.

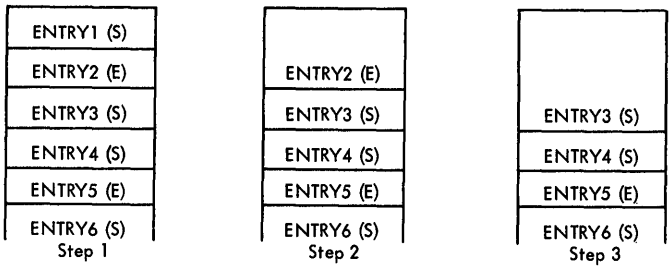


Figure 6. ENQ Macro Instruction Processing

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.

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 the performance of a task which has not been assigned control of the resource. Both of these abnormal termination conditions can be avoided by either 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, 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=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" situation occurs when two or more tasks are dependent on each other in such a way that none of the tasks can be taken out of the wait condition until one of the same tasks has been performed. An example of a fully developed interlock situation is shown in Figure 7. The task represented by ENTRY 1 in List 1 is the same task represented by ENTRY 2 in List 2. The task represented by ENTRY 2 in List 1 is the same task represented by ENTRY 1 in List 2. Control of the resource represented by List 1 is assigned to the task which is waiting for the resource represented by List 2. Control of the resource represented by List 2 is assigned to the task which is waiting for the resource represented by List 1. 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.

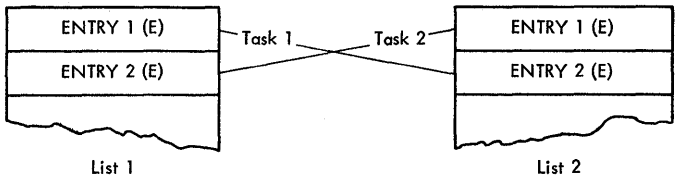


Figure 7. Interlock Condition

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 7 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; control of any of the resources will not be given until control of all resources requested in the macro instruction can be given. For example, instead of coding the two ENQ macro instructions shown in Example 20, the one ENQ macro instruction shown in Example 21 could be coded. If all requests were made in this manner, it would avoid the interlock shown in Figure 7. 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.

```
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,     C
      NAME3ADD,NAME4ADD,E,10,SYSTEM)
```

Example 21. One Request for Two Resources

- If the use of one resource always depends on the use of a second resource, then the pair of resources can be defined as one resource in the ENQ and DEQ macro instructions. This procedure can be used for any number of resources that are always used in 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 the information from the task control block. The full power of the EXTRACT macro instruction is available (and needed) only in an operating system with MVT or with 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 task control block for the active task or any of its subtasks. The following information can be requested:

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

- 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 task input/output table. This is the only information provided in response to an EXTRACT macro instruction when using an operating system with the primary control program or MFT.

You must provide an area into which the control program places the information you request. If you request all of the fields (by coding FIELDS=ALL), the area must be seven full words 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. If you request information other than the address of the task input/output table when you are using an operating system with PCP or 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 a sign character 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.

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 the primary control program or MFT, only one time interval can be in effect at any one time during the job step. With an operating system with MVT, 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 micro-seconds (1 timer unit).
- DINTVL - requires an 8-byte field containing unpacked decimal digits of the form HHMMSSth (hours, minutes, seconds, tentns 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 micro-seconds. If the interval has already expired, the content of 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

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. WAIT should not be coded in an operating system with the primary control program, because no productive work can be performed when the only task is in a wait 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 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.

```
         ...
         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, go through loop again
         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

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 publication IBM System/360 Operating System: Supervisor and Data Management Macro Instructions.

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 publication IBM System/360

Operating System: Supervisor and Data Management Macro Instructions.

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.

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

Example 24. Writing to the Operator

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 not have the MCS option. In an operating system with PCP, the reply is available when, following execution of the WTOR macro instruction, your program regains control. But 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, anotner 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 not receive the original message.

WRITING TO THE PROGRAMMER

The WTO and the WTOR macro instructions allow you to write messages to the programmer, as well as to the operator.

At system generation (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 publication IBM System/360 Operating System: Job Control Language Reference.) 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

```
         ...
         WTOR   'STANDARD OPERATING CONDITIONS?  REPLY YES OR NO',          C
                REPLY,3,ECBAD,ROUTCDE=(1,15),DESC=7
         WAIT   ECB=ECBAD
         ...
ECBAD    DC     Event control block
REPLY    DC     Answer area'    '
```

Example 25. Writing to the Operator With a Reply

the message output class, as described above. The message can also go to the console(s) in the situations described by Table 6.

- Table 6. Using WTO and WTOR to Write Messages to the Programmer

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.		

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.

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.
- The system log is supported, but the system log data sets are temporarily inactive because both are full and waiting to be written.

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

Although when using the WTL macro instruction you code the message within apostrophes, the written message does not contain the apostrophes. The message can include any character that is valid for the 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 8, 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.

```
         ...
         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

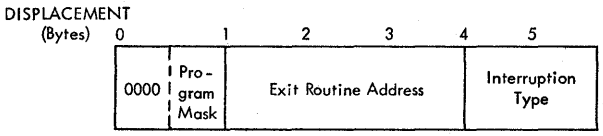


Figure 8. Program Interruption Control Area

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 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 9.

The PICA address in the program interruption element is the address of the program interruption control area used in

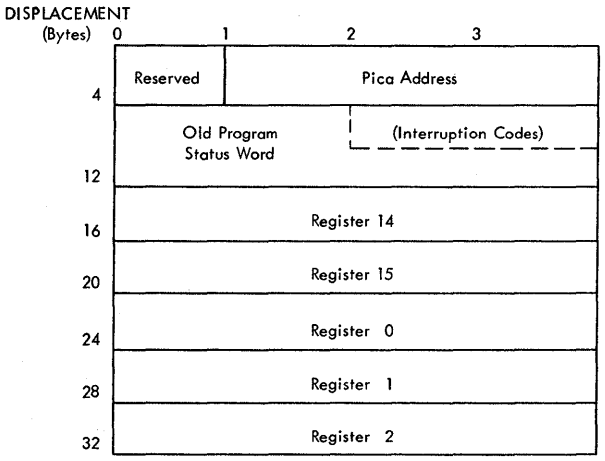


Figure 9. Program Interruption Element

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:

- 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, and 195, 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. Table 7 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. Table 8 defines the conditions under which interruptions are precise in Models 65-195. Note that interruptions are always precise in systems with lower model numbers.

• Table 7. Interruption Code in the Old Program Status Word

Type of Exception	Type of Interruption							
	Precise (ILC ≠ 0)		Imprecise (ILC = 0)					
	All Models		Models 65–85		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)	0101	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)
Significance	(zero)	1110			000000001000	(zero)	000000001000	(zero)
Floating-point Divide	(zero)	1111			000000000100	(zero)	000000000100	(zero)

Interruptions in the Models 91 and 195: As shown in Table 7, 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.

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.

• Table 8. Precise Interruptions in IBM System/360 Models 65, 67, 75, 85, 91, and 195

Type of Exception	Models 65–85		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 publication IBM System/360 Model 91 Functional Characteristics.

[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 publication IBM System/360 Model 195 Functional Characteristics.

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 Table 8. 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.

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 10 is an example of one of these. The routine shown in Figure 10 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

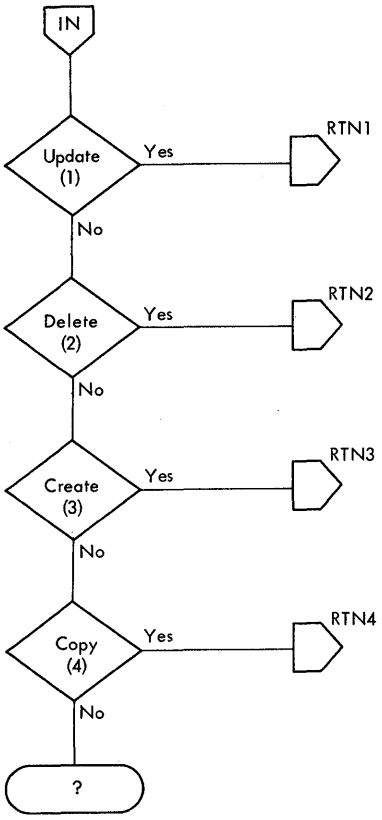


Figure 10. Abnormal Condition Detection

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.

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 causes only that task and the subtasks of that task to

be abnormally terminated. 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 a system with PCP, MFT, or MVT, 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.

INTERCEPTION OF ABNORMAL TERMINATION

Abnormal termination of a task can be intercepted through the use of the STAE macro instruction. When an ABEND macro instruction is scheduled for a task that has previously issued a STAE macro instruction, the ABEND macro instruction 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 the facility of scheduling a STAE retry routine, see the publication IBM System/360 Operating System: System Programmer's Guide.

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

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

The user can also issue a STAE macro instruction to cancel (make the previous STAE request active) or to 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 zero.

When a program using STAE returns control to a previous level via an SVC 3, all STAE requests are canceled. If a STAE request specifies the "XCTL=YES" option, that STAE request is not canceled when the STAE user issues an XCTL macro instruction. If a program terminates by any means other than an SVC 3, all STAE requests must be canceled by the terminating program before returning control to another program.

Example 27 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. 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", but the parameter list address and the "XCTL=YES" request remain the same.

```
        ...
        STAE  EXIT1,CT,PARAM=LIST1,XCTL=YES  Initial STAE request.
        ...
        ...
        LA    5,EXIT2                       Put new exit routine address in register 5.
        STAE  (5),OV                        STAE request to overlay exit routine address.
        ...
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 27. Use of the STAE Macro Instruction

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

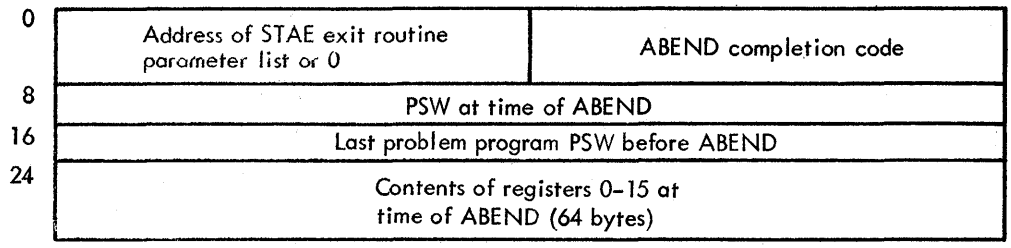
- Registers 0, 1: Unpredictable.
- Registers 2-13: Same as when STAE was issued.
- Register 14: Unpredictable.
- Register 15: Error/completion code.

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.

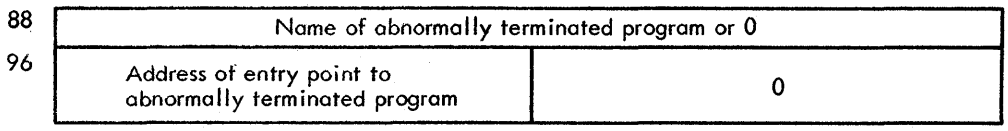
When a program with an active STAE environment encounters an ABEND situation, control will be returned to the user at the STAE exit routine address. However, if the abnormal termination is caused by either an operator's CANCEL, job step timer expiration, or the detaching of an incomplete task, ABEND processing continues, and the STAE exit routine is not executed. At this time, active I/O for the failing task either has been quiesced and is restorable at a later time, or has been halted and is not restorable. 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.



If a problem program issued STAE:



If supervisor program issued STAE:

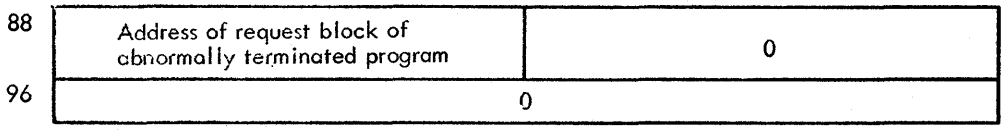


Figure 11. Work Area for STAE Exit Routine

- Register 1: Address of a 104-byte work area, as shown in Figure 11.
- 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.

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
- Register 1: ABEND completion code 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).

- Register 2: Address of STAE exit parameter list.

- Register 3-13: Unpredictable.
- Register 14: Return address.
- Register 15: Exit routine address.

Upon completion of the STAE exit routine, the user must indicate whether ABEND processing is to be continued for the task or whether a STAE retry routine should be scheduled. The return codes to be placed in register 15 are defined as follows:

Code	Indication
0	ABEND processing is to continue.
4	A retry routine has been provided and the Request Block chain should be purged.
8	A retry routine has been provided and the Request Block chain should not be purged.

For further information on the option of STAE retry, see the publication IBM System/360 Operating System: System Programmer's Guide.

THE DUMP

There are two ways in which dumps of main storage can be obtained: through the use of the DUMP operand in the ABEND macro instruction and through the use of the SNAP macro instruction. 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 IBM System/360 Operating System: 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 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.

REQUIREMENTS

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 publication IBM System/360 Operating System: 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 PCP and MVT. (The data control block is discussed in Section II of this manual.)
- 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 publication IBM System/360 Operating System: Storage Estimates for storage requirements.

INDICATIVE DUMP

In an operating system with the primary control program or MFT, you can obtain an indicative dump, as shown in the publication IBM System/360 Operating System: 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.

Main Storage Management

No matter which configuration of the operating system you are using, there is a finite amount of main storage available to your job step. If 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. You should remember the following requirements when using the primary control program if your job is ever going to be run in an operating system with MFT or MVT.

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 publication IBM System/360 Operating System: 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 double word 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 active task 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 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 28. 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 28. 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

```
             ...
             GETMAIN  EC,LV=16000,A=ANSWADD,     Conditional request for 16000 bytes
                      HIARCHY=0                  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,     Attempt to get smaller amount
                      HIARCHY=0                  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'                    Minimum size for procedure 1
SIZES        DC       F'4000'                    Minimum 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 28. Use of the GETMAIN Macro Instruction

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 was 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 PCP SYSTEMS AND IN MFT SYSTEMS WITHOUT SUBTASKING)

There is only one unnumbered subpool in an operating system with the primary control program or MFT. In these configurations 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 12 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 can share subpool 0, Request 1 and Request 2 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.

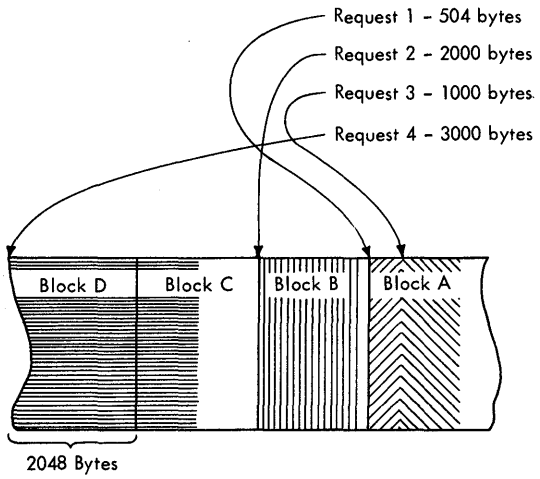


Figure 12. Main Storage Control

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 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 REQUEST

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 the primary control program or 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 due to a parity error, a fresh copy can be loaded to overlay the copy in main storage, and execution can be resumed. If the module is designated as "refreshable" when processed by the linkage editor, a fresh copy is loaded automatically by the machine check handler. The machine check handler, available with MFT or MVT, is optional programming support with Model 65, and standard programming support with Model 65 Multiprocessor and Model 85; it is not available with the primary control program.

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 publication IBM System/360 Operating System: Supervisor and Data Management Macro Instructions 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:

```
┌--------┬-----┬---------------------------┐
|[symbol]|POINT|dcb address,block address  |
└--------┴-----┴---------------------------┘
```

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 publication IBM System/360 Operating System: Supervisor and Data Management Macro Instructions. The expansion of the standard form of these macro instructions (that is, the form described in Section II of that publication) results in an inline 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.

```
          ...
          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 29. Using the List and the Execute Forms of the DEQ Macro Instruction

Example 29 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.

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 publication IBM System/360 Operating System: Linkage Editor and Loader. 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 under the primary control program, MFT, or MVT, 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 a PCP or 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 publication IBM System/360 Operating System: Supervisor and Date Management Macro Instructions.

Checkpoint and Restart

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.

With the primary control program, MFT, or MVT, 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 30 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 30. Establishing a Checkpoint

```
          ...
          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 31. Canceling a Request for Automatic Restart

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 31.

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 32.

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 UCS buffer for a 1403 printer with the Universal Character Set feature. The buffer contents are not saved when a checkpoint is taken. To reload the buffer upon restart, you must reissue the SETPRT macro instruction in the

```
          ...
          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 32. Obtaining Updated TCB Information After Restart

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

Example 33. Requesting a Resource After Restart

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 33 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 publication, IBM System/360 Operating System: 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.
- 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 not discussed in detail until Section II of this publication, but it is one of the most important considerations in selecting checkpoints. 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 publication, IBM System/360 Operating System: Job Control Language Reference.

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 PCP, the data set may be on a unit record device or on magnetic tape. 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 34. 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 IBM System/360 Operating System: 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 34. 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 35 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 35, is described in "Modifying the Data Control Block," Section II, Part 1.)

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 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.

```
         ...
         DCBD   DSORG=PS              Define IHADCB (dummy section that defines
         CSECT                          DCBDDNAM)
         ...
         LA     2,CHECKDCB            Establish CHECKDCB as base address
         USING  IHADCB,2                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 35. Alternating Use of Checkpoint Data Sets

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

Example 36. Assigning a Checkpoint Identification

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 36 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 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 37, 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.

```
         ...
         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=U,BLKSIZE=32760,                   C
               DDNAME=CHKDD
```

Example 37. Recording a Checkpoint Identification Assigned by the Control Program

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 (primary control program, 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. Therefore, with the primary control program, the main storage size for the alternate system must be at least as large as that for the original system.
- 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 publication IBM System/360 Operating System: Job Control Language Reference.

Section II: Data Management Services

Section II describes the data management features and facilities of the operating system. The reader should be familiar with the theory and philosophy of System/360 Operating System data management and with the various general terms and concepts necessary to begin preparation for actual coding. Each macro instruction is discussed in sufficient detail so that the reader can turn directly to the macro instruction format description to determine the operand requirements. Format descriptions are in the publication IBM System/360 Operating System: Supervisor and Data Management Macro Instructions.

Part 1, Introduction to Data Management, is concerned with the characteristics of data sets and direct access devices. It also describes the means and methods used to communicate with the operating system during program assembly and execution. It contains a general description of the various control blocks, their contents, and their functions.

Part 2, Data Management Processing Procedures, describes data access and processing techniques in terms of data set organization, buffer acquisition and control, and jobs to be done. The major emphasis is on work requirements rather than access methods.

Part 3, Data Set Disposition and Space Allocation, describes the techniques required for efficient and effective data set disposition and space allocation. A sufficiently detailed description of the data definition (DD) statement is included to get the reader "on-the-air."

Data Set Characteristics

The manner in which data is transferred between main storage and external devices is of paramount importance in most data processing applications. The data management function of the System/360 Operating System assists you in achieving maximum efficiency in managing the mass of data associated with the many programs that are processed at an installation. To attain this objective, data management facilities have been designed to provide systematic and effective means of organizing, identifying, storing, cataloging, and retrieving all data, including loadable programs, processed by the operating system.

Data set storage control, supported by an extensive catalog system, makes it possible for you to retrieve data by symbolic name alone, without specifying device types and volume serial numbers. In freeing computer personnel from the necessity of maintaining involved volume serial number inventory lists of data and programs stored within the system, the catalog reduces manual intervention and the possibility of human error.

Data sets stored within the cataloging system can be classified according to installation needs. For example, a sales department could classify the data it uses by geographic area, by individual salesman, or by any other logical plan.

The cataloging system also makes it possible for you to classify successive generations or updates of related data. These generations can be given an identical name and subsequently be referred to relative to the current generation. The system automatically maintains a list of the most recent generations.

Data from a direct access volume, a remote terminal, or a tape, and data organized sequentially or as in a library, may be requested by you in essentially the same way. In addition, data management provides:

- Allocation of space on direct access volumes. Flexibility and efficiency of direct access devices is improved through greater use of available space.
- Automatic retrieval of data sets by name alone.
- Freedom to defer specifications such as buffer length, block size, and device type until the job is submitted for processing. This permits the creation of programs that are in many ways independent of their operating environment.

Control of confidential data is provided by the data set security facility of the System/360 Operating System. Using this facility, you can prevent unauthorized access to payroll data, sales forecast data, and all other data sets requiring special security attention. The security-protected data set is available for processing only when a correct password is furnished.

The data access facilities provided by the operating system are a major extension of previous input/output control systems. Input/output routines are provided to efficiently schedule and control the transfer of data between main storage and input/output devices. Routines are available to:

- Read data.
- Write data.
- Block and deblock records.
- Overlap reading, writing, and processing operations.
- Read and verify volume and data set labels.
- Write data set labels.
- Automatically position and reposition volumes.
- Detect error conditions and correct when possible.
- Provide exits to user-written error and label routines.

Corresponding to the range of system facilities available for control of data is an equal range of facilities for access to the data. The variety of techniques for gaining access to a data set is derived from two variables: data set organization and data access technique.

Operating System/360 data sets can be organized in four ways:

- Sequential: This is the familiar tape-like structure, in which records are placed in physical rather than logical sequence. Thus, given one record, the location of the next record is determined by its physical position in the data set. The sequential organization is used for all magnetic tapes, and may be selected for direct access devices. Punched tape, punched cards, and printed output are considered to be sequentially organized.

- Indexed Sequential: Records are arranged in collating sequence, according to a key that is a part of every record, on the tracks of a direct access volume. In addition, a separate index or set of indexes maintained by the system gives the location of certain principal records. This permits direct as well as sequential access to any record.

- Direct: This organization is available for data sets on direct access volumes. The records within the data set may be organized in any manner you choose. All space allocated to the data set is available for data records. No space is required for indexes. Records are stored and retrieved directly with addressing specified by you.

- Partitioned: This structure has characteristics of both the sequential and the indexed sequential organizations. Independent groups of sequentially organized data, called members, are in direct access storage. Each member has a simple name stored in a directory that is part of the data set and contains the location of the member's starting point. Partitioned data sets are generally used to store programs. As a result, they are often referred to as libraries.

Requests for input/output operations on data sets through macro instructions are divided into two categories or techniques: the technique for queued access and the technique for basic access. Each technique is identified according to its treatment of buffering and input/output synchronization with processing. The combination of an access technique and a given data set organization is called an access method. In choosing an access method for a data set, therefore, you must consider not only its organization, but also the macro instruction capabilities. Also, you may choose a data organization according to the access techniques and processing capabilities available. The code generated by the macro instructions for both techniques is optionally reenterable depending on the form in which parameters are expressed.

In addition to the access methods provided by the operating system, an elementary access technique called execute channel program is also provided. To use this technique, you must establish your own system for

organizing, storing, and retrieving data. Its primary advantage is the complete flexibility it allows you in using the computing facilities directly.

An important feature of data management is that much of the detailed information needed to store and retrieve data, such as device type, buffer processing technique, and format of output records need not be supplied until the job is ready to be executed. This device independence permits changes to be made in those details without requiring changes in the program. Therefore, you may design and test a program without knowing the exact input/output devices that will be used when it is executed.

Device independence is a feature of both access techniques when you are processing a sequential data set. The degree of device independence achieved is to some extent determined by you. Many useful device-dependent features are available as part of special macro instructions, and achieving device independence requires some selectivity in their use.

DATA SET IDENTIFICATION

Any information that is a named, organized collection of logically related records can be classified as a data set. The information is not restricted to a specific type, purpose, or storage medium. A data set may be, for example, a source program, a library of macro instructions, or a file of data records used by a processing program.

Whenever you indicate that a new data set is to be created and placed on auxiliary storage, you (or the operating system) must give the data set a name. The data set name identifies a group of records as a data set. All data sets recognized by name (i.e., referred to without volume identification) and all data sets residing on a given volume must be distinguished from one another by unique names. To assist in this, the system provides a means of qualifying data set names.

A data set name is one simple name or a series of simple names joined together so that each represents a level of qualification. For example, the data set name DEPT58.SMITH.DATA3 is composed of three simple names that are delimited to indicate a hierarchy of categories. Proceeding from the left, each simple name is a category within which the next simple name is a subcategory.

Each simple name consists of one to eight alphameric characters, the first of which must be alphabetic. The special character period (.) separates simple names from each other. Including all simple names and periods, the length of the data set name must not exceed 44 characters. Thus, a maximum of 22 qualification levels is possible for a data set name.

To permit different data sets to be processed without program reassembly, the data set is not referred to by name in the processing program. When the program is executed, the data set name and other pertinent information (e.g., unit type and volume serial number) is specified in a job control statement called the data definition (DD) statement. To gain access to the data set during processing, a reference is made to a data control block associated with the name of the DD statement. Space for a data control block is reserved by a DCB macro instruction when your program is assembled.

DATA SET STORAGE

System/360 provides a variety of devices for collecting, storing, and distributing data. Despite the variety, the devices have many common characteristics. For convenience, therefore, the generic term volume is used to refer to a standard unit of auxiliary storage. A volume may be any one of the following:

- A reel of magnetic tape.
- A disk pack.
- A bin in a data cell.
- A drum.
- That part of an IBM 2302 disk storage device served by one access mechanism (the device would have either two or four volumes in all).

Each data set stored on a volume has its name, location, organization, and other control information stored in the data set label or volume table of contents (direct access volumes only). Thus, when the name of the data set and the volume on which it is stored are made known to the operating system, a complete description of the data set, including its location on the volume, can be retrieved. Following this, the data itself can be retrieved, or new data added to the data set.

Various groups of labels are used in secondary storage of the System/360 Operating System to identify magnetic tape and direct access volumes, as well as the data sets they contain. Magnetic tape volumes can have standard or nonstandard labels, or they can be unlabeled. Direct access volumes must use standard labels. Standard label support includes a volume label, a data set label for each data set, and optional user labels.

Keeping track of the volume on which a particular data set resides can be a burden and often a source of error. To alleviate this problem, the system provides for automatic cataloging of data sets. A cataloged data set can be retrieved by the system if given only the name of the data set. If the name is qualified, each qualifier corresponds to one of the indexes in the catalog. For example, the data set DEPT58.SMITH.DATA3 is found by searching a master index to determine the location of the index name DEPT58. That index is then searched to find the location of the index SMITH. Finally, that index is searched for DATA3 to find the identification of the volume containing the required data set.

By use of the catalog, collections of data sets related by a common external name and the time sequence in which they were cataloged (i.e., their generation) can be identified, and are called generation data groups. For example, a data set name LAB.PAYROLL(0) refers to the most recent data set of the group; LAB.PAYROLL(-1) refers to the second most recent data set, etc. The same collection of data set names can be used repeatedly -- with no requirement to keep track of the volume serial numbers used.

DIRECT ACCESS VOLUMES

Direct access volumes play a major role in the System/360 Operating System. They are used to store executable programs, including the operating system itself. Direct access storage is also used for data and for temporary working storage. One direct access storage volume may be used for many different data sets, and space on it may be reallocated and reused. A volume table of contents (VTOC) is used to account for each data set and available space on the volume.

Each direct access volume is identified by a volume label, which is usually stored in track 0 of cylinder 0. You may specify up to seven

additional labels for further identification. These are located after the standard volume label.

The volume table of contents describes the contents of the direct access volume. It is a data set that is composed of a series of data set control blocks (DSCB), each of which is composed of one or more control blocks. The VTOC can contain the following data set control blocks:

- A DSCB for each data set on the volume.
- A DSCB that indicates the space allocated to the VTOC itself.
- A DSCB for all tracks on the volume that are available for allocation.

The DSCB for each data set contains the name, description, and location of the data set on the volume. Its size depends on the organization and the number of noncontiguous areas of the data set.

Each direct access volume is initialized by a utility program before being used on the system. The initialization program generates the proper volume label and constructs the table of contents. For additional information on direct access labels, see Appendix A.

When a data set is to be stored on a direct access volume, you must supply the operating system with control information designating the amount of space to be allocated to the data set. The amount of space can be expressed in terms of blocks, tracks, or cylinders. Space can be allocated in a device-independent manner if the request is expressed in terms of blocks. If the request is made in terms of tracks or cylinders, you must be aware of such device considerations as cylinder capacity and track size.

MAGNETIC TAPE VOLUMES

Because of the sequential organization of magnetic tape devices, the operating system does not require space allocation facilities comparable to those for direct access devices. When a new data set is to be placed on a magnetic tape volume, you must specify the data set sequence number if it is not the first data set on the reel. A volume with standard labels or no labels will be positioned by the operating system so that the data set can be read or written. If the data set has nonstandard labels, the installation must provide volume-positioning in its nonstandard label processing routines. All data sets stored on a given magnetic tape volume must be recorded in the same density.

When a data set is to be stored on an unlabeled tape volume and you have not specified a volume serial number, the system assigns a serial number to that volume and to any additional volumes required for the data set. Each such volume is assigned a serial number of the form Lxxxyy where xxx will indicate the data set sequence number from IPL to IPL and yy will indicate the volume sequence number for the data set. If you specify volume serial numbers for unlabeled volumes on which a data set is to be stored, the system assigns volume serial numbers to any additional volumes required. If data sets residing on unlabeled volumes are to be cataloged or passed, you should specify the volume serial numbers for the volumes required. This will prevent data sets residing on different volumes from being cataloged or passed under identical volume serial numbers. Retrieval of such data sets could result in unpredictable errors.

Each data set and each data set label group on magnetic tape that is to be processed by the operating system must be followed by a tapemark. Tapemarks cannot exist within a data set. When the operating system is used to create a tape with standard labels or no labels, all tapemarks

are automatically written. Two tapemarks are written following the last trailer label group on a volume to indicate the last data set on the volume. On an unlabeled volume, the two tapemarks are written following the last data set.

When the operating system is used to create a tape data set with nonstandard labels, the delimiting tapemarks are not written. If the data set is to be retrieved by the operating system, those tapemarks must be written by an appropriate installation nonstandard label processing routine. Otherwise, tapemarks are not required following nonstandard labels since positioning of the tape volumes must be handled by the installation routines.

For more information on labels for magnetic tape volumes, refer to the publication IBM System/360 Operating System: Tape Labels.

DATA SET RECORD FORMATS

A data set is composed of a collection of records that usually have some logical relation to one another. The record is the basic unit of information used by a processing program. It might be a single character, all information resulting from a given business transaction, or parameters recorded at a given point in an experiment. Much data processing consists of reading, processing, and writing individual records.

The process of grouping a number of records before writing them on a volume is referred to as blocking. A block is considered to be made up of the data between interrecord gaps (IRG). Each block can consist of one or more records. Blocking conserves storage space on the volume because it reduces the number of interrecord gaps in the data set. In many cases, blocking also increases processing efficiency by reducing the number of input/output operations required to process a data set.

Records may be in one of three formats: fixed-length (format F), variable-length (format V), or undefined-length (format U). The prime consideration in the selection of a record format is the nature of the data set itself. You must know the type of input your program will receive and the type of output it will produce. Selection of a record format is based on this knowledge, as well as on an understanding of the type of input/output devices that are used to contain the data set and the access method used to read and write the data records. The record format of a data set is indicated in the data control block according to specifications in the DCB macro instruction, the DD statement, or the data set label.

Note: There is no minimum requirement for block size; however, if a data check occurs on a magnetic tape device, any records shorter than 12 bytes in a read operation or 18 bytes in a write operation will be treated as a noise record and lost. No check for noise will be made unless a data check occurs.

FIXED-LENGTH RECORDS

The size of fixed-length (format F) records, shown in Figure 13, is constant for all records in the data set. The number of records within a block is usually constant for every block in the data set, unless the data set contains truncated (short) blocks. If the data set contains unblocked format F records, one record constitutes one block.

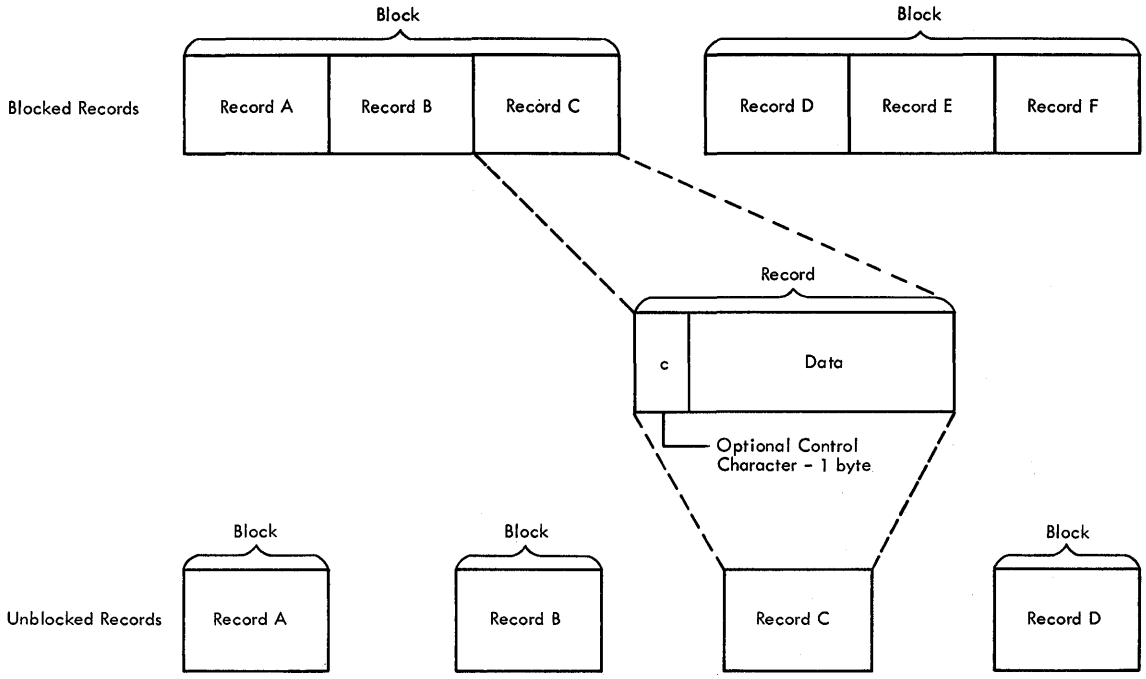


Figure 13. Fixed-Length Records

The system automatically performs physical length checking on blocked format F records, making allowance for truncated blocks. Because the channel and interrupt system can be used to accommodate length checking, and the blocking/deblocking is based on a constant record length, format F records can be processed faster than format V.

A sequential data set is said to contain records in standard format F if:

- All records in the data set are format F.
- Every track except the last is filled to capacity (no room for another record).
- No blocks except the last are truncated.

Standard format F data sets can be read from direct access storage more efficiently than data sets with truncated blocks because the system can determine the location of each block to be read. If you use standard format F records to create a sequential data set in direct access storage, the system puts the same number of blocks on each track.

Format F records are shown in Figure 13. The optional control character (C), used for stacker selection or carriage control, may be included in each record to be printed or punched.

VARIABLE-LENGTH RECORDS

Format V provides for (1) variable-length records, (2) variable-length record segments, each of which describes its own characteristics, and (3) variable-length blocks of such records or record segments. The control program performs length checking of the block and uses the record or segment length information in blocking and

deblocking. The first four bytes of each record, record segment, or block are a descriptor word containing control information. You must allow for these additional four bytes in both your input and output buffers.

Block Descriptor Word: A variable-length block consists of a block descriptor word (BDW) followed by one or more logical records or record segments. The block descriptor word is a four-byte field which describes the block. The first two bytes specify the block length ('LL') -- four bytes for the BDW plus the total length of all records or segments within the block. This length must be in the range $8 \leq LL \leq 32,760$ or, when using WRITE with tape, $18 \leq LL \leq 32,760$. The third and fourth bytes are reserved for future system use and must be zero. If the system does your blocking -- that is, when you use the queued access technique -- the operating system automatically provides the BDW when it writes the data set. If you do your own blocking -- that is, when you use the basic access technique -- you must supply the BDW.

Record Descriptor Word: A variable-length logical record consists of a record descriptor word (RDW) followed by the data. The record descriptor word is a four-byte field describing the record. The first two bytes contain the length ('*ll*') of the logical record (including the four-byte RDW). The length must be in the range $4 \leq ll \leq 32,756$. All bits of the third and fourth bytes must be zero as other values are used for spanned records. For output, you must provide the RDW. For input, the operating system provides the RDW except in data mode (spanned records). In data mode, the system passes the record length to the user in the logical record length field (DCBLRECL) of the data control block. The optional control character (C) may be specified as the fifth byte of each record and must be followed by at least one byte of data. The RDW and the control character, if specified, are not punched or printed.

Figure 14 shows blocked and unblocked variable-length records without the spanning feature.

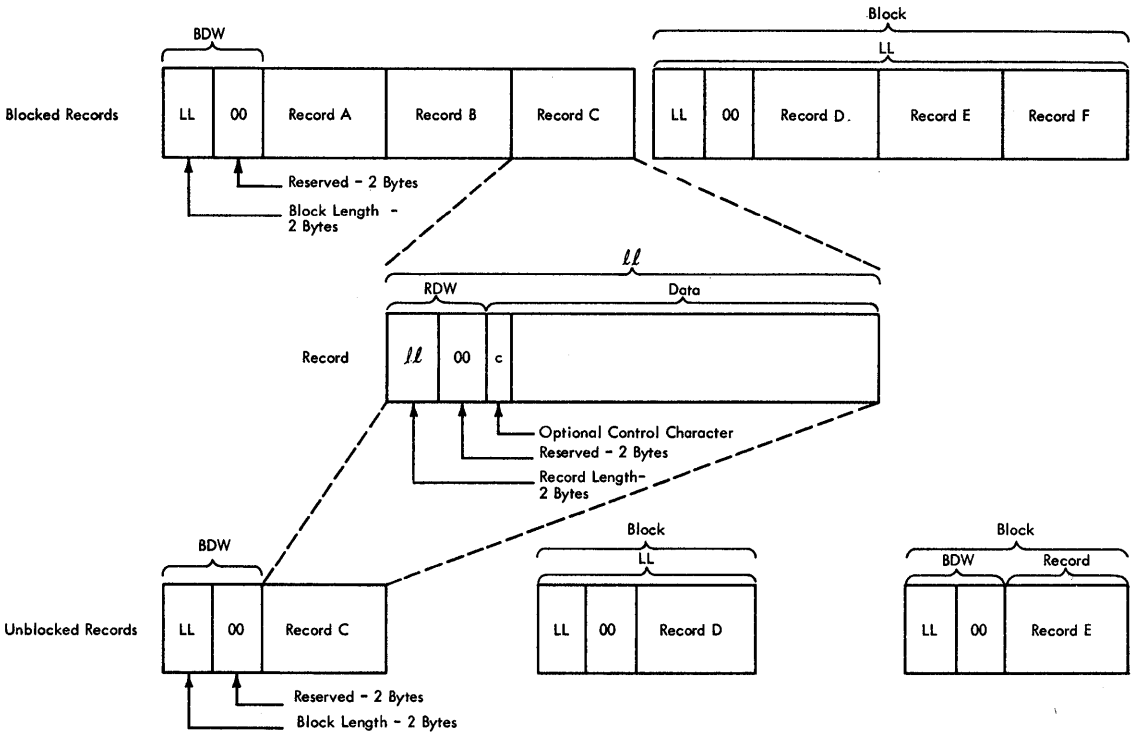
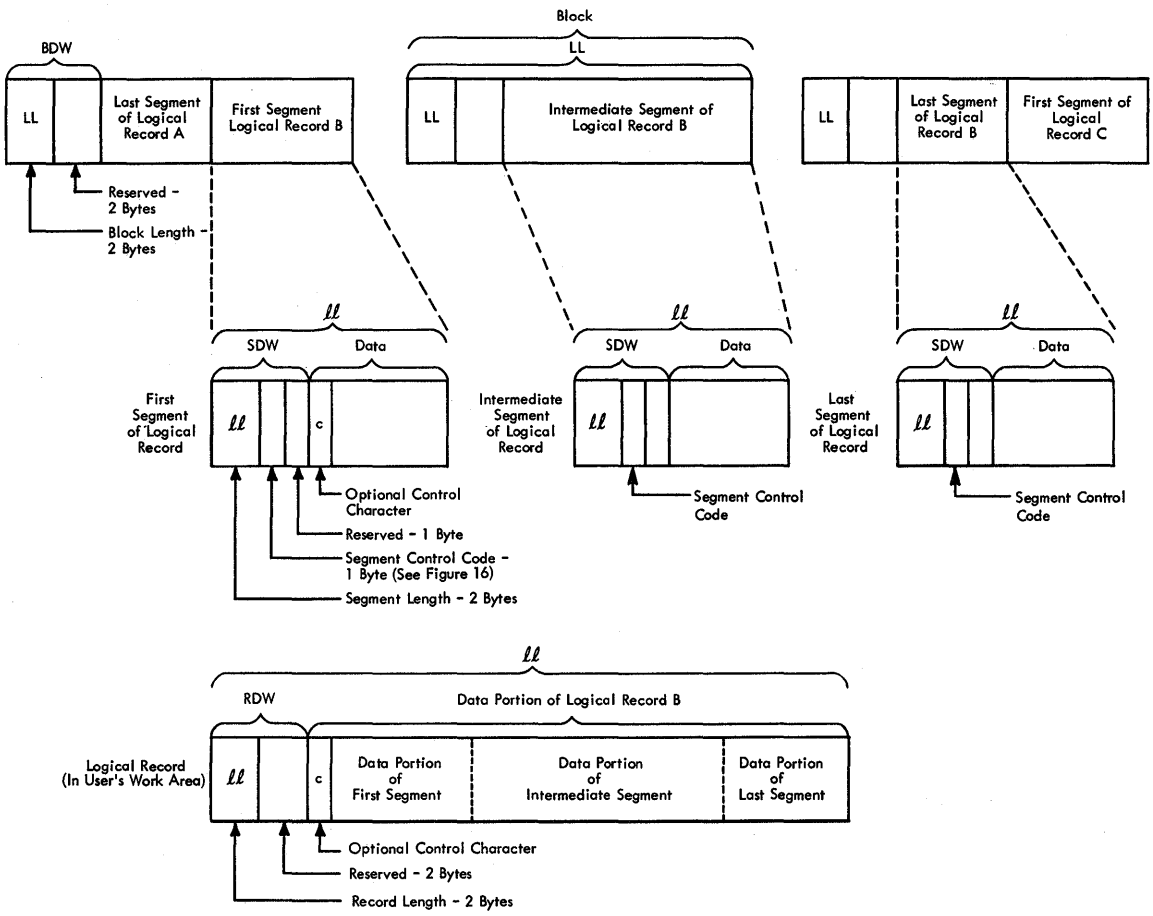


Figure 14. Variable-Length Records

SPANNED VARIABLE-LENGTH RECORDS (SEQUENTIAL ACCESS METHODS): The spanning feature of the queued and basic sequential access methods enables the user to create and process variable-length logical records which are larger than one physical block and/or to pack blocks with variable-length records. This is done by splitting the records into segments so that they can be written into more than one block, as shown in Figure 15.



Note: Not All Segment and Block Combinations are Represented

Figure 15. Spanned Variable-Length Records

When spanning is specified for blocked records, the system tries to fill all blocks. For unblocked records, a record which is larger than block size is split and written in two or more blocks -- each block containing only one record or record segment. Thus the block size may be set to the one which is best for a given device or processing situation. It is not restricted by the maximum record length of a data set. A record may, therefore, span several blocks, and may even span volumes. (Note that a logical record spanning three or more volumes cannot be processed in update mode using QSAM.) A block can contain a combination of records and record segments but not multiple segments of the same record. When records are added to or deleted from a data set, or when the data set is processed again with different block- or record-size parameters, the record segmenting will change.

Segment Descriptor Word: Each record segment consists of a segment descriptor word (SDW) followed by the data. The segment descriptor word, similar to the record descriptor word, is a four-byte field which describes the segment. The first two bytes contain the length ('*ll*') of the segment including the four-byte SDW. The length must be in the range $4 \leq ll \leq 32,756$ or, when using WRITE with tape, $18 \leq ll \leq 32,756$. The third byte of the SDW contains the segment control code, which specifies the relative position of the segment in the logical record. The segment control code is in the rightmost two bits of the byte. The segment control codes are shown in Figure 16. The remaining bits of the third

byte and all of the fourth byte are reserved for future system use and must be zero.

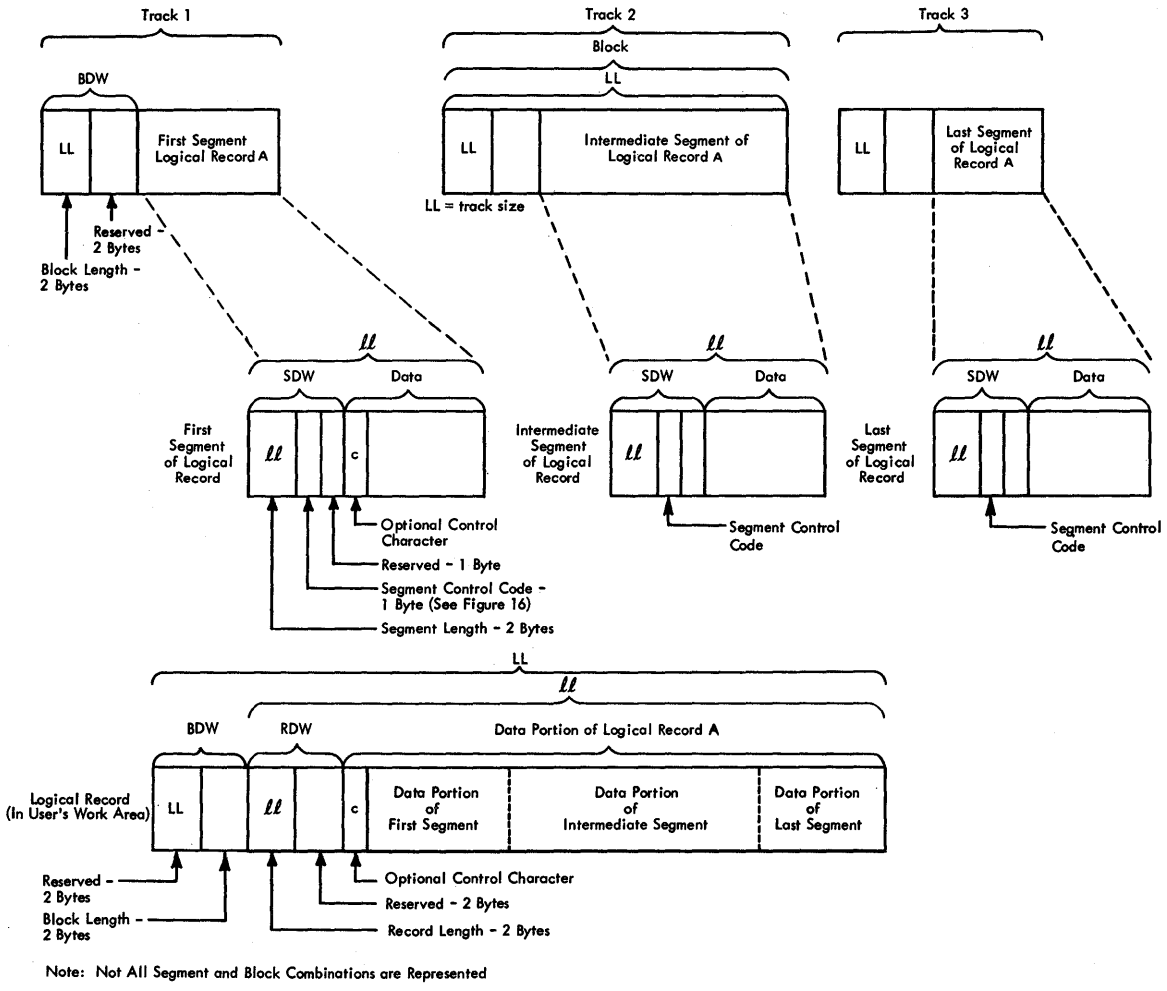
Binary Code	Relative Position of Segment
00	Complete logical record.
01	First segment of a multi-segment record.
10	Last segment of a multi-segment record.
11	Neither first nor last segment of a multi-segment record.

Figure 16. Segment Control Codes

The SDW for the first segment replaces the RDW for the record after the record has been segmented. The SDW may be built by the user or the system depending on which mode of processing is used. In the basic sequential access method, the user must create and interpret the spanned records himself. In the queued sequential access method move mode, complete logical records, including the RDW, are processed in the user's work area. GET consolidates segments into logical records and creates the RDW. PUT forms segments as required and creates the SDW for each segment. Data mode is similar to move mode but allows reference only to the data portion of the logical record in the user's work area. The logical record length is passed to the user through the DCBLRECL field of the data control block. In the locate mode, both GET and PUT process one segment at a time. However, in the locate mode, if the user provides his own record area using the BUILDRCD macro instruction or if he asks the system to provide his record area by specifying BFTEK=A, GET, PUT, and PUTX process one logical record at a time. A logical record spanning three or more volumes cannot be processed when the data set is opened for update.

When unit record devices are used with spanned records, the system assumes unblocked records and the block size must be equivalent to one print line or one card. Records which span blocks are written one segment at a time.

SPANNED VARIABLE-LENGTH RECORDS (BASIC DIRECT ACCESS METHOD): The spanning feature of the basic direct access method enables the user to create and process variable-length unblocked logical records that are longer than one track. The feature also enables the user to pack tracks with variable-length records by splitting the records into segments. These segments can then be written onto more than one track, as shown in Figure 17.



• Figure 17. Spanned Variable-Length Records for BDAM Data Sets

When you specify spanned, unblocked record format for the basic direct access method and when a complete logical record cannot fit on the track, the system tries to fill the track with a record segment. This aspect of spanning means that the maximum record length of a data set is not restricted by block size. Furthermore, it allows a record to span several tracks, with each segment of the record on a different track. However, since the system does not allow a record to span volumes, all segments of a logical record in a direct data set are on the same volume.

UNDEFINED-LENGTH RECORDS

Format U is provided to permit processing of any records that do not conform to the F or V formats. As shown in Figure 18, each block is treated as a record; therefore, deblocking must be performed by your program. The optional control character may be used in the first byte of each record. Because the system does not perform length checking on format U records, your program may be designed to read less tnan a complete block into main storage.

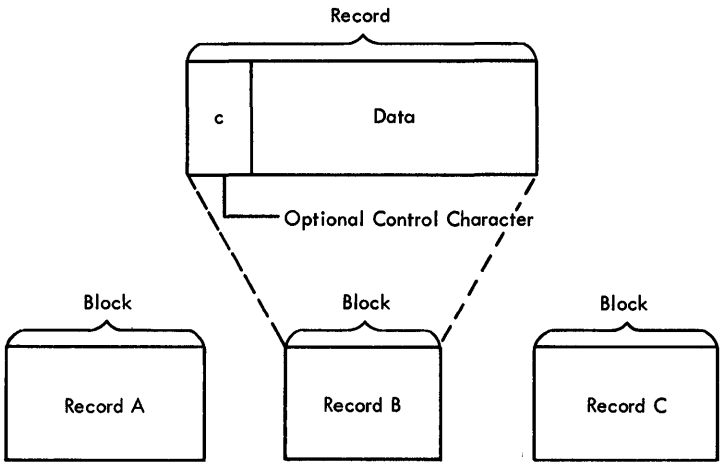


Figure 18. Undefined-Length Records

CONTROL CHARACTER

You may specify in the DD statement, the DCB macro instruction, or the data set label that an optional control character is part of each record in the data set. The one-byte character is used to indicate a carriage control channel when the data set is printed or a stacker bin when the data set is punched. Although the character is a part of the record in storage, it is never printed or punched. For that reason, buffer areas must be large enough to accommodate the character. If the immediate destination of the record is a device that does not recognize the control character, e.g., disk, the system assumes that the control character is the first byte of the data portion of the record. If the destination of the record is a printer or punch and you have not indicated the presence of a control character, the system regards it as the first byte of data. A list of the control characters is in Appendix B.

Direct Access Device Characteristics

Regardless of organization, data sets created using the operating system can be stored on a direct access volume. Each block has a distinct location and a unique address making it possible to locate any record without extensive searching. Thus, records can be stored and retrieved either directly or sequentially.

Although direct access devices differ in physical appearance, capacity, and speed, they are functionally and logically similar in terms of data recording, checking, format, and programming. The recording surface of each volume is divided into many tracks, each defined as the circumference of the recording surface. The tracks are arranged concentrically; their number and capacity varies with the device. Each device has some type of access mechanism, containing a number of read/write heads that transfer data as the recording surface rotates past.

The logical arrangement of related tracks is vertical rather than horizontal. As shown in Figure 19, a 2311 cylinder is comprised of ten tracks, which is equal to the number of recording surfaces. Because there are 203 tracks per disk, there are 203 vertical cylinders of ten tracks each.

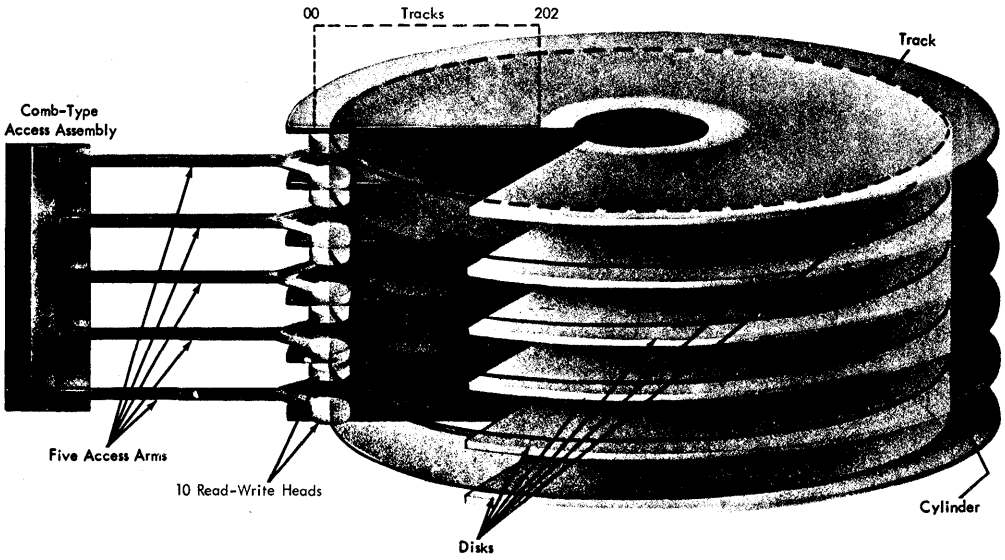


Figure 19. 2311 Disk Drive

TRACK FORMAT

Information is recorded on all direct access volumes in a standard format. In addition to device data, each track contains a track descriptor record ("capacity record" or R0), and data records. As shown in Figure 20, there are two possible data record formats -- Count-Data and Count-Key-Data -- only one of which can be used for a particular data set.

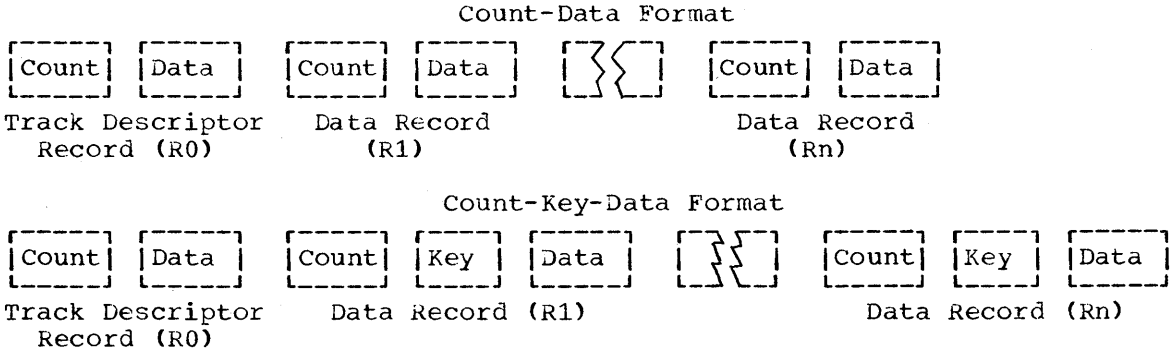


Figure 20. Direct Access Volume Track Formats

In addition to device data, the count area contains eight bytes that identify the location of the record in terms of the cylinder, head, and record numbers; its key length (0 if no keys are used); and its data length.

If the records are written with keys, the key area (1 to 255 bytes) contains a record key that identifies the following data area in terms of a part number, account number, sequence number, etc. In some cases, records are written with keys so that they can be located quickly.

TRACK ADDRESSING

There are two types of addresses that can be used to store and retrieve data on a direct access volume: actual and relative. The only real advantage of using actual addresses is the reduction in time required to convert from relative to actual address and vice versa. When sequentially processing a multiple volume data set, you can refer only to records of the current volume.

Actual Addresses: When the system returns the actual address of a record on a direct access volume to your program, it is in the form MBBCCHHR, where:

M
: is a one-byte binary number specifying the relative location of an entry in a data extent block (DEB). The data extent block is created by the system when the data set is opened. Each extent entry describes a set of contiguous tracks allocated for the data set.

BBCCHH
: is three two-byte binary numbers specifying the cell (bin), cylinder, and head number for the record, i.e., its track address. The cylinder and head numbers are recorded in the count area for each record.

R
: is a one-byte binary number specifying the relative block number on the track. The block number is also recorded in the count area.

If you use actual addresses in your program, the data set must be treated as "unmovable."

Relative Addresses: There are two kinds of relative addresses that can be used to refer to records in a direct access data set: relative block address or relative track address.

The relative block address is provided as a three-byte binary number that indicates the position of the block in relation to the first block of the data set. Allocation of noncontiguous tracks does not affect the number. Therefore, the first block of a data set always has a relative block address of zero.

The relative track address is provided in the form TTR, where:

TT
: is a two-byte binary number specifying the position of the track in relation to the first track allocated for the data set. The TT for the first track is zero. Allocation of noncontiguous tracks does not affect the number.

R
: is a one-byte binary number specifying the number of the block in relation to the first block on the track TT. The first block of data on a track has a record value of one.

TRACK OVERFLOW

If the record overflow feature is available for the direct access device being used, you can reduce the amount of unused space on the volume by specifying the track overflow option in the DD statement or the DCB macro instruction associated with the data set. If the option is used, a block that does not fit on the track is partially written on that track and continued on the next available track. Each segment of

an overflow block (the portion written on one track) has a count area. The data length field in the count area specifies the length of that segment only. If the block is written with a key, there is only one key area for the block. It is written with the first segment. If the option is not used, blocks are not split between tracks.

Although a block can begin on one track and continue on the next, it cannot be continued on a noncontiguous track or from one separately allocated area to another.

WRITE VALIDITY CHECK

You can specify the write validity option in either the DD statement or the DCB macro instruction. The system will read each record back (without data transfer) and, by testing for a data check from the I/O device, verify that the record transferred from main to secondary storage was written correctly. This verification requires an additional revolution of the device for each record tnat was written. Standard error recovery procedures are initiated if an error condition is detected.

Interface with the Operating System

You must describe the characteristics of a data set, the volume on which it resides, and its processing requirements before processing can begin. During execution, the descriptive information is made available to the operating system in the data control block. A data control block is required for each data set, and is created in a processing program by a DCB macro instruction.

Primary sources of information to be placed in the data control block are a DCB macro instruction, a data definition (DD) statement, or a data set label. In addition, you can provide or modify some of the information during execution by storing the pertinent data in the appropriate field of the data control block. The specifications needed for input/output operations are supplied during the initialization procedures of the OPEN macro instruction. Therefore, the pertinent data can be provided when your job is to be executed rather than when you write your program (see Figure 21).

When the OPEN macro instruction is executed, the open routine performs four primary functions:

- Completes the data control block.
- Loads all necessary data access routines not already in main storage.
- Initializes data sets by reading or writing labels and control information.
- Constructs the necessary system control blocks.

Information from a DD statement is stored in the job file control block (JFCB) by the operating system. When the job is to be executed, the JFCB is made available to the open routine. The data control block is filled by using information from the DCB macro instruction, the JFCB, or an existing data set label. If more than one source specifies a particular field, only one source is used. A DD statement takes precedence over a data set label; a DCB macro instruction over both. However, you can modify any data control block field either before the data set is opened, or when control is returned to your program by the operating system (during the data control block exit). Some fields can be modified during processing.

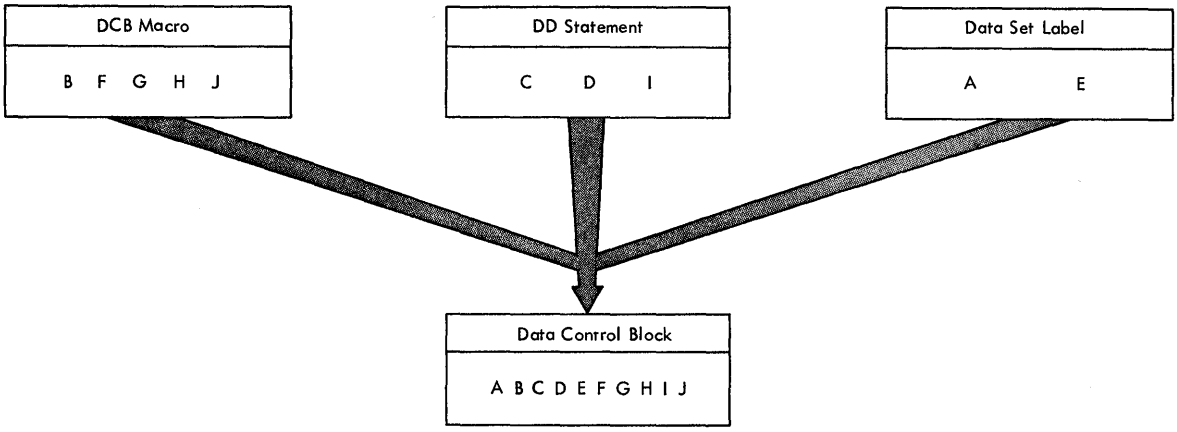


Figure 21. Completing the Data Control Block

Figure 22 illustrates the process and the sequence of filling in the data control block from various sources. The primary source is your program, i.e., the DCB macro instruction. In general, you should use only those DCB parameters that are needed to ensure correct processing. The other parameters can be filled in when your program is to be executed. When a data set is opened, any field in the JFCB not completed by a DD statement is filled in from the data set label (if one exists). Any field not completed in the data control block is filled in from the JFCB. Any field in the data control block then can be completed or modified by your own DCB exit routine.

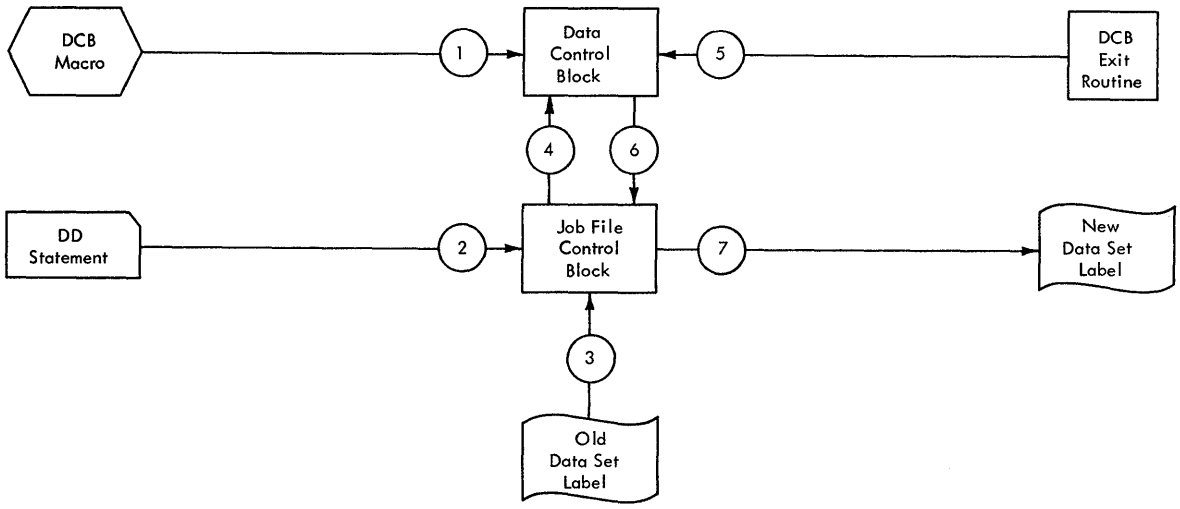


Figure 22. Source and Sequence for Completing the Data Control Block

When the data set is closed, the data control block is returned to its condition prior to opening; it is then available for reuse with another data set. The open and close routines also use the updated JFCB to write the data set labels for output data sets. If the data set is not closed when your job terminates, the operating system will perform the close functions automatically.

There is usually one data control block for each data set. It is possible to concurrently open more than one data control block for processing the same data set on a direct access volume. However, you must exercise caution with respect to volume positioning, switching, space allocation, label processing, and device control.

DATA SET DESCRIPTION

For each data set you are going to process there must be a corresponding data control block and data definition statement. The characteristics of the data set and device-dependent information can be supplied by either source. In addition, the DD statement must supply data set identification, device characteristics, space allocation requests, and related information. The logical connection between a data control block and a DD statement is made by specifying the name of the DD statement in the DCB macro instruction (DDNAME), or by completing the field yourself before opening the data set.

Once the data set characteristics have been specified in the DCB macro instruction, they can only be changed by modifying the data control block during execution. The fields of the data control block discussed below are common to most data organizations and access techniques.

Data Set Organization (DSORG): specifies the organization of the data set as physical sequential (PS), indexed sequential (IS), partitioned (PO), or direct (DA). If the data set contains location-dependent information (i.e., absolute rather than relative addresses), it must be marked as unmovable, e.g., PSU. You must specify the data set organization in the DCB macro instruction. When creating an indexed sequential or direct organization data set, this information must also be supplied in the DD statement.

Record Format (RECFM): specifies the characteristics of the records in the data set as fixed-length (F), variable-length (V), or undefined-length (U). Blocked records are specified as being FB or VB. Track overflow can be requested, e.g., FBT.

Record Length (LRECL): specifies the length, in bytes, of each record in the data set. If the records are variable-length, the maximum record length must be specified. For input, the field should be omitted for format U records.

Block Size (BLKSIZE): specifies the maximum length, in bytes, of a block. If the records are format F, the block size must be an integral multiple of the record length, including the key length, except for SYSOUT data sets. (See "Writing a SYSOUT Data Set" in Section II, Part 3, of this book.) If the records are format V, the block size must be the maximum block size. If records are unblocked, the block size must be four bytes greater than the record length (LRECL). When spanned variable-length records are specified, the block size is independent of the record length (LRECL).

Each of the data set description fields of the data control block, except as noted for data set organization, can be specified when your job is to be executed. In addition, data set identification and disposition, as well as device characteristics, can be specified at that time. The parameters of the DD statement discussed below are common to most data set organizations and devices.

Data Definition Name (DDNAME): is the name of the DD statement and provides a logical relationship to the data control block that specifies the same ddname.

Data Set Name (DSNAME): specifies the name of a newly defined data set, or refers to a previously defined data set.

Data Control Block (DCB): provides, by means of subparameters, information to be used to complete those fields of the data control block that were not specified in the DCB macro instruction. This parameter cannot be used to modify a data control block.

Channel Separation and Affinity (SEP/AFF): requests that specified data sets use different channels during input/output operations.

Input/Output Device (UNIT): specifies the quantity and type of I/O devices to be allocated for use by the data set.

Space Allocation (SPACE): designates the amount of space on a direct access volume that should be allocated for the data set. Unused space can be released when your job is finished.

Volume Identification (VOLUME): identifies the particular volume or volumes, or the number of volumes to be assigned to the data set or the volumes on which existing data sets reside.

Data Set Label (LABEL): indicates the type and contents of the label or labels associated with the data set. The operating system verifies standard labels (SL) or standard user labels (SUL). Nonstandard labels (NSL) can be specified only if your installation has incorporated into the operating system routines to write and process nonstandard labels.

Data Set Disposition (DISP): describes the status of a data set and indicates what is to be done with it at the end of the job step.

PROCESSING PROGRAM DESCRIPTION

There are several types of processing information required by the operating system to ensure proper control of your input/output operations. The forms of macro instructions in the program, buffering requirements, and the addresses of your special processing routines must be specified during either the assembly or the execution of your program. The DCB parameters specifying buffer requirements are discussed in the section "Buffer Acquisition and Control."

Because macro instructions are expanded during the assembly of your program, you must supply the macro instruction forms that are to be used in processing each data set in the associated DCB macro instruction. Buffering requirements and related information can be supplied in the DCB macro instruction, the DD statement, or by storing the pertinent data in the appropriate field of the data control block before the end of your DCB exit routine. If the addresses of special processing routines are omitted from the DCB macro instruction, you must complete them in the data control block before opening the data set.

Macro Instruction Form (MACRF): specifies not only the macro instructions used in your program, but also the processing mode as discussed in the section "Buffer Control." The organization of your data set, the macro instruction form, and the processing mode determine which of the data access routines will be used during execution.

Exits to Special Processing Routines: The DCB macro instruction can be used to identify the location of:

- A routine that performs end-of-data procedures.
- A routine that supplements the operating system's error recovery routine.
- A list that contains addresses of special exit routines.

The exit addresses can be specified in the DCB macro instruction or you can complete the data control block fields before opening the data set. Table 9 summarizes the exits that you can specify either explicitly in the data control block, or implicitly by specifying the address of an exit list in the data control block.

Table 9. Data Management Exit Routines

Exit Routine	When Available	Where Specified
End-of-Data-Set	No more sequential records or blocks are available	EODAD operand
Error Analysis	After an uncorrectable input/output error	SYNAD operand
Standard User Label (physically sequential or direct organization.)	Opening and closing or reaching the end of a data set, and when changing volumes.	EXLST operand and exit list
Data Control Block	Opening a data set	EXLST operand and exit list
End-of-Volume	When changing volumes	EXLST operand and exit list
Block Count	After unequal block count compare by EOV	EXLST operand and exit list

End-of-Data-Set Exit Routine (EODAD): specifies the address of your end-of-data routine that performs any final processing on an input data set. This routine is entered when a READ or GET request is made and there are no more records or blocks to be retrieved. (On a READ request, your routine is entered when you issue a CHECK macro instruction to check for completion of the read operation.) Your routine can reposition the volume for continued processing (BPAM only), close the data set, or process the next sequential data set. Under no condition should you issue another GET request after the data set has encountered the end-of-data condition (QSAM only). If no routine is provided, the task will be abnormally terminated.

Synchronous Error Routine Exit (SYNAD): specifies the address of an error routine that is to be given control when an input/output error occurs. This routine can be used to analyze exceptional conditions or uncorrectable errors. The error can be skipped, accepted, or processing can be terminated.

If an input/output error occurs during data transmission, standard error recovery procedures, provided by the operating system, attempt to correct the error before returning control to your program. An uncorrectable error usually causes an abnormal termination of the task. However, if you specify in the DCB macro instruction the address of an error analysis routine, the routine is given control in the event of an uncorrectable error.

You can write a SYNAD routine to determine the cause and type of error that occurred by examining:

- The contents of the general registers.
- The data event control block (discussed in Part 2).
- The exceptional condition code.
- The standard status and sense indicators.

There is a special macro instruction, SYNADAF, that you can use to perform this function automatically. This macro instruction produces a

descriptive error message that can be printed by a subsequent PUT or WRITE macro instruction.

Having completed the analysis, you can return control to the operating system or close the erroneous data set and terminate processing. In no case can you attempt to reread or rewrite the record since the system has already attempted to recover from the error.

When using GET/PUT macro instructions to process a sequential data set, the operating system provides three automatic error options (EROPT) to be used if there is no SYNAD routine or if you want to return control to your program from the SYNAD routine:

- ACC -- accept the erroneous block.
- SKP -- skip the erroneous block.
- ABE -- abnormally terminate the task.

These options are applicable only to data errors, as control errors will result in abnormal termination of the task. Data errors affect only the validity of a block of data. Control errors affect information or operations necessary for continued processing of the data set. These options are not applicable to output errors, with the exception of output errors on the printer. When chained scheduling is used, the SKP option is not available, and defaults to the ACC option if coded. If the EROPT and SYNAD fields are not completed, ABE is assumed.

Upon entry to your SYNAD routine register 0 will contain either the address of standard status indicators and a displacement value to reach the channel command word (GET/PUT), or the address of the data event control block (READ/WRITE). Register 1 indicates which macro instruction caused the error and the address of the data control block. Registers 2 through 13 remain as they were. Register 14 contains a return address and 15 the address of your SYNAD routine.

Your SYNAD routine can end by branching to another routine in your program, such as a routine that closes the data set. It can also end by returning control to the control program, which then returns control to the next sequential instruction (after GET, PUT, etc.) in your program. If your routine returns control, the conventions for saving and restoring registers are as follows:

- The SYNAD routine must preserve the contents of registers 13 and 14. If required by the logic of your program, the routine must also preserve the contents of registers 2 through 12. Upon return to your program, the contents of registers 2 through 12 will be the same as upon return to the control program from the SYNAD routine.

- The SYNAD routine must not use the save area whose address is in register 13, because this area is used by the control program. If the routine saves and restores registers, it must provide its own save area.

- If the SYNAD routine calls another routine or issues supervisor or data management macro instructions, it must provide a save area in the usual way or by means of a SYNADAF macro instruction. The SYNADAF macro instruction provides a save area for its own use, and then makes this area available to the SYNAD routine. Such a save area must be removed from the save area chain by issuing a SYNADRLS macro instruction before returning control to the control program.

When you use READ/WRITE macro instructions, errors are detected when you issue a CHECK macro instruction. If you are processing a direct or sequential data set and you return directly to the CHECK routine from your SYNAD routine, the operating system regards that as an acceptance of the bad record. If you are creating a direct data set and you return

to the CHECK routine from your SYNAD routine, your task will be abnormally terminated.

When you use QSAM to read and translate paper tape characters, your SYNAD routine receives control when you request the record preceding the record in error. However, before giving control to your SYNAD routine, the system translates the requested record into your buffer.

More specifically, suppose that you are using QSAM to read and translate a paper tape data set and that you have specified in your DCB SYNAD=(address) and EROPT=ACC. Suppose also that the third record of the data set has a parity error. When you issue a GET request for the second record, the system translates that record into your buffer and, as a result of the error in the third record, passes control to your SYNAD routine. Because you specified the accept option, the system returns control to your program after your SYNAD error analysis routine completes its processing. When you issue a GET request for the third record, that record is translated into your buffer as follows:

- The system translates the characters, up to the character in error, into your buffer.
- The system moves the character in error into your buffer without translating it.
- The system translates the remaining characters of the record into your buffer.

Exit List (EXLST): specifies the address of special processing routines. An exit list must be created if user label, data control block, end-of-volume, or block count exits are used.

The exit list is constructed of four-byte entries that must be aligned on fullword boundaries. The exit routine type is specified by a code in the high-order byte, and the address of the routine is specified in the three low-order bytes. Codes and addresses for the exit routines are shown in Table 10.

You can activate or deactivate any entry in the list by placing the required code in the high-order byte. Care must be taken, however, so as not to destroy the last entry indication. The list will be scanned from top to bottom by the operating system. The first active entry found with the proper code will be selected.

• Table 10. Format and Contents of an Exit List

Routine Type	Hexadecimal Code	3-Byte Routine Address - Purpose
Inactive entry	00	Ignored; the entry is not active
Input header label	01	Process a user input header label
Output header label	02	Create a user output header label
Input trailer label	03	Process a user input trailer label
Output trailer label	04	Create a user output trailer label
Data control block exit	05	Data control block exit routine
End-of-volume	06	End-of-volume exit routine
User totaling	0A	Pointer to user's totaling area
Block count exit	0B	Block count unequal exit routine
Defer input trailer label	0C	Defer processing of a user input trailer label from EOD until CLOSE
Defer nonstandard input trailer label	0D	Defer processing a nonstandard input trailer label on magnetic tape unit from EOD until CLOSE (no exit routine address)
Last entry	80	Last entry in list. This code can be specified with any of the above but must always be specified with the last entry.

The list can be shortened during execution by setting the high-order four bits to the hexadecimal value 8. The list can be extended by setting the high-order four bits to zero.

When control is passed to an exit routine, the general registers contain the following information:

Register	Contents
0	Variable; see exit routine description.
1	Address of data control block currently being processed.
2-13	Contents prior to execution of the macro instruction.
14	Return address (must not be altered by the exit routine).
15	Address of exit routine entry point.

The conventions for saving and restoring registers are as follows:

- The exit routine must preserve the contents of register 14. It need not preserve the contents of other registers. The control program restores registers 2-13 before returning control to your program.
- The exit routine must not use the save area whose address is in register 13, because this area is used by the control program. If the exit routine calls another routine or issues supervisor or data management macro instructions, it must provide the address of a new save area in register 13.

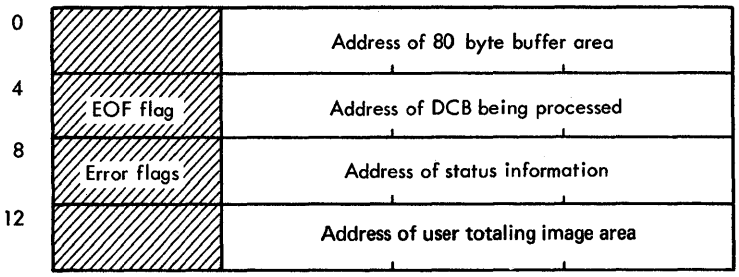
Standard User Label Exit: When you create a data set with physically sequential or direct organization, you can provide routines to create your own data set labels. You can also provide routines to verify these labels when you use the data set as input. The labels are 80 characters long with the first four characters UHL1,UHL2,..., UHL8 for header labels or UTL1,UTL2,..., UTL8 for trailer labels.

The physical location of the labels on the data set depends on the data set organization. For direct data sets (using BDAM), user labels are placed on a separate user label track in the first volume. User label exits are taken only during OPEN and CLOSE. Thus you may create or examine up to eight user header labels only during OPEN and up to eight trailer labels only during CLOSE. Since the trailer labels are on the same track as the header labels, the first volume of the data set must be mounted when the data set is closed. For physically sequential data sets (using BSAM or QSAM), you may create or examine up to eight header labels and eight trailer labels on each volume of the data set. The user label exits are taken during OPEN, CLOSE, and EOV processing.

To create or verify labels, you must specify the addresses of your label exit routines in an exit list for use during standard label processing. Thus you may have separate routines for creating or verifying header and trailer label groups. Care must be taken if a magnetic tape is read backwards since the trailer label group is processed as header labels and the header label group is processed as trailer labels.

When your routine receives control, the contents of general register 0 are unpredictable. Register 1 contains the address of a parameter list. The contents of registers 2-13 are the same as when the macro instruction was issued. However, if your program does not issue the CLOSE macro instruction, or abnormally terminates before issuing CLOSE, the CLOSE macro instruction will be issued by the control program, with control program information in these registers.

The parameter list pointed to by register 1 is a 16-byte area aligned on a fullword boundary and contains the following:



The first address in the parameter list points to an 80-byte label buffer area. For input, the control program reads a user label into this area before passing control to the label routine. For output, the user label routine constructs labels in this area and returns to the control program, which writes the label. When an input trailer label routine receives control, the EOF flag (high-order byte of the second entry in the parameter list) will be as follows:

bit 0 = 0: Entered at end-of-volume.
bit 0 = 1: Entered at end-of-file.
bits 1-7 : Reserved.

When a user label exit routine receives control after an uncorrectable I/O error has occurred, the third entry of the parameter

list contains the address of the standard status information. The error flag (high-order byte of the third entry in the parameter list) will be as follows:

```
bit 0 = 1   : Uncorrectable I/O error.
bit 1 = 1   : Error occurred when writing updated label.
bits 2-7    : Reserved.
```

The fourth entry in the parameter list is the address of the user totaling image area. This image area is the entry in the user totaling save area which corresponds to the last record physically written on the volume. The image area is discussed further under "User Totaling."

Each routine must create or verify one label of a header or trailer label group, place a return code in register 15, and return control to the operating system. The operating system responds to the decimal return code as shown in Table 11.

Table 11. System Response to a User Label Exit Routine Return Code

Routine Type	Return Code	System Response
Input header or trailer label	0	Normal processing is resumed. If there are any remaining labels in the label group, they are ignored.
	4	The next user label is read into the label buffer area and control is returned to the exit routine. If there are no more labels in the label group, normal processing is resumed.
	8	The label is written from the label buffer area and normal processing is resumed.*
	12	The label is written from the label area, then the next label is read into the label buffer area and control is returned to the label processing routine. If there are no more labels, processing is resumed.*
Output header or trailer label	0	Normal processing is resumed; no label is written from the label buffer area.
	4	User label is written from the label buffer area. Normal processing is resumed.
	8	User label is written from the label buffer area. If less than eight labels have been created, control is returned to the exit routine, which then creates the next label. If eight labels have been created, normal processing is resumed.
*Only for a physically sequential data set opened for UPDATE or a direct data set opened for UPDATE or OUTPUT.		

You can create user labels only for data sets on direct access or magnetic tape volumes with standard labels. When you specify standard

and user labels in a DD statement (LABEL=SUL) and there is an active entry in the exit list, a label exit is always taken. A label exit may be taken when an input data set does not contain user labels, or when no user label track has been allocated for writing labels on a direct access volume. In either case, the appropriate exit routine is entered with the buffer area address parameter set to zero. On return from the exit routine, normal processing is resumed; no return code is necessary.

Label exits are not taken for system output (SYSOUT) data sets, or for data sets on volumes that do not have standard labels. For other data sets, exits are taken as follows:

- When the data set is opened, header label exits are taken, except when the data set already exists and DISP=MOD is coded in the DD statement. In the latter case, the volume is positioned to the end of the data set, and input trailer label exits are taken.

- When end-of-volume is reached, trailer label exits are taken; header label exits are taken after volume switching. Input trailer label exits are not taken, however, if you force end-of-volume by issuing an FEOV macro instruction.

- When end-of-data is reached, input trailer label exits are taken before the EODAD exit, unless the data control block (DCB) exit list indicates defer input trailer label processing. When an output data set is closed, output trailer label exits are taken.

- When end-of-data is reached for a direct access data set and the data control block (DCB) exit list indicates defer input trailer label processing, the system changes the 0C to 0D. When the Close routine has finished processing, the system changes the code back to 0C.

To process records in reverse order, a data set on magnetic tape can be read backwards. When you read backwards, header label exits are taken to process trailer labels, and trailer label exits are taken to process header labels. The system presents labels from a label group in ascending order by label number, which is the order in which the labels were created. If necessary, an exit routine can determine label type (UHL or UTL) and number (1 to 8) by examining the first four characters of each label.

If an uncorrectable error occurs while reading or writing a user label, the system passes control to the appropriate exit routine with the third word of the parameter list flagged and pointing to status information.

After an input error, the exit routine must return control with an appropriate return code (0 or 4). No return code is required after an output error. If an output error occurs while the system is opening a data set, the data set is not opened (DCB is flagged) and control is returned to your program. If an output error occurs at any other time, the system attempts to resume normal processing.

A sample program illustrating user label processing is included in SYS1.SAMPLIB. This program, named USERLABL, is discussed in the publication, IBM System/360 Operating System: System Generation.

User Totaling: (BSAM, QSAM only) When creating or processing a data set with user labels, you may develop control totals for each volume of the data set and store this information in your user labels. For example, a control total that was accumulated as the data set was created can be stored in your user label and later compared with a total accumulated while processing the volume. The user totaling

facility assists you by synchronizing the control data which you create with records physically written on a volume. For an output data set without user labels, you can also develop a control total which will be available to your end-of-volume routine.

To request this facility, you must specify OPTCD=T in the DCB macro instruction or in the DCB parameter of the DD statement. The area in which you accumulate the control data, the user's totaling area, must be identified to the control program by an X'0A' entry in the data control block (DCB) exit list.

The user's totaling area, an area in storage that you provide, must begin on a halfword boundary and be large enough to contain your accumulated data plus a two-byte length field. The length field must be the first two bytes of the area and specify the length of the entire area. A data set for which you have specified user totaling (OPTCD=T) will not be opened if either the totaling area length or the address in the exit list is zero, or if there is no X'0A' entry in the exit list.

The control program establishes a user totaling save area, in which the control program preserves an image of your totaling area, when an I/O operation is scheduled. When the output user label exits are taken, the address of the save area entry (user totaling image area) corresponding to the last record physically written on a volume is passed to you in the fourth entry in the User Label parameter list. This parameter list is described in the section "Standard User Label Exit." When an end-of-volume exit is taken for an output data set and user totaling has been specified, the address of the user totaling image area is in register 0.

When using this facility for an output data set, i.e., when creating the data set, you must update your control data in your totaling area prior to issuing a PUT or a WRITE macro instruction. The control program places an image of your totaling area in the user totaling save area when an I/O operation is scheduled. A pointer to the save area entry (user totaling image area) corresponding to the last record physically written on the volume, is presented to you in your label processing routine. Thus, you can include the control total in your user labels. When subsequently using this data set for input, you can accumulate the same information as you read each record and compare this total with the one previously stored in the user trailer label. If you have stored the total from the preceding volume in the user header label of the current volume, you can process each volume of a multi-volume data set independently and still maintain this sytem of control.

When variable-length records are specified with the totaling facility for user labels, special considerations are necessary. Since the control program determines whether a variable-length record will fit in a buffer after a PUT or a WRITE has been issued, the total you have accumulated may include one more record than is actually written on the volume. In the case of variable-length spanned records, the accumulated total will include the control data from the volume-spanning record although only a segment of the record is on that volume. However, when processing such a data set, the volume-spanning record or the first record on the next volume will not be available to you until after the volume switch and user label processing is completed. Thus the totaling information in the user label may not agree with that developed while processing the volume. One way you can resolve this situation is to maintain, when you are creating a data set, control data pertaining to each of the last two records and include both totals in your user labels. Then the total related to the last complete record on the volume and the volume-spanning record or the first record on the next volume would be

available to your user label routines. During subsequent processing of the data set, your user label routines can determine if there is agreement between the generated information and one of the two totals previously saved.

Data Control Block Exit: You can specify in an exit list the address of a routine that completes or modifies a data control block and does any additional processing required before the data set is completely open. The routine is entered during the opening process after the job file control block has been used to supply information for the data control block. The routine can be used to determine data set characteristics by examining fields completed by the data set labels.

As with label processing routines, register 14 must be preserved and restored if any macro instructions are used in the routine. Control is returned to the operating system by a RETURN macro instruction; no return code is required.

End-of-Volume Exit: You can specify in an exit list the address of a routine that is entered when end-of-volume is reached in processing a physically sequential data set.

When the end-of-volume routine is entered, register 0 contains zero unless user totaling was specified. If you specified user totaling in the DCB macro instruction (OPTCD=T) or in the DD statement for an output data set, register 0 will contain the address of the user totaling image area. The routine is entered after a new volume has been mounted and all necessary label processing has been completed. If the volume is a reel of magnetic tape, the tape is positioned after the tapemark that precedes the beginning of the data.

The end-of-volume exit routine can be used to take a checkpoint by issuing the CHKPT macro instruction, which is discussed in "Section 1: Supervisor Services". If the job step terminates abnormally, it can be restarted from this checkpoint. When the job step is restarted, the volume is mounted and positioned as upon entry to the routine. Note that restart becomes impossible if changes are subsequently made to the system SVC library (SYS1.SVCLIB). When the step is restarted, pointers to end-of-volume modules must be the same as when the checkpoint was taken.

The end-of-volume exit routine returns control in the same manner as the data control block exit routine. Register 14 must be preserved and restored if any macro instructions are used in the routine. Control is returned to the operating system by a RETURN macro instruction; no return code is required.

Block Count Exit: You can specify in an exit list the address of a routine that will allow you to abnormally terminate the task or continue processing when the end-of-volume routine finds an unequal block count condition. When using standard label input tapes, the block count in the trailer label is compared by end-of-volume with the block count in the data control block. The count in the trailer label reflects the number of blocks written when the data set was created. The number of blocks read when the tape is used as input is contained in the DCBBLKCT field of the data control block.

The routine is entered during end-of-volume processing. The trailer label block count will be passed in register 0. The user may access the count field in the data control block by addressing the address passed in register 1 plus the proper displacement as given in IBM System/360 Operating System: System Control Blocks. If the block count in the data control block differs from that in the trailer label when no exit routine is provided, the task is abnormally terminated.

The routine must terminate with a RETURN macro instruction and a return code that indicates what action is to be taken by the operating system as shown in Table 12. As with other exit routines, register 14 must be saved and restored if any macro instructions are used.

Table 12. System Response to Block Count Exit Return Code

Return Code	System Action
0	The task is abnormally terminated.
4	Normal processing is resumed.

Defer Nonstandard Input Trailer Label Exit: In an exit list, you can specify a code that indicates that you want to defer nonstandard input trailer label processing from end-of-data time until close time. The address portion of the entry is not used by the operating system.

An end-of-volume condition exists in several situations. Two are when the system reads a filemark or tapemark at the end of a volume of a multivolume data set but that volume is not the last, and when the system reads a filemark or tapemark at the end of a data set. The first situation is referred to here as an end-of-volume condition, the second, as an end-of-data condition, although it, too, can occur at the end of a volume.

For an end-of-volume condition, the EOV routine will pass control to the user's nonstandard input trailer label routine, whether or not this exit code is specified. For an end-of-data condition when this exit code is specified, the EOV routine does not pass control to the user's nonstandard input trailer label routine. Instead, the Close routine passes control to the user's routine.

MODIFYING THE DATA CONTROL BLOCK

You can complete or modify the data control block during execution of your program. You can also determine data set characteristics from information supplied by the data set labels. Changes or additions can be made prior to opening the data set, after closing it, during the DCB exit routine, or while the data set is open. Naturally, any information must be supplied before it is needed.

Because each data control block does not have a symbolic name for each field, a DCBD macro instruction must be used to supply the symbolic names. By loading a base register with the address of the data control block to be processed, any field can be referred to symbolically.

The DCBD macro instruction generates a dummy control section (DSECT) named IHADCB. The name of each field begins with DCB followed by the first five letters of the keyword operand that represents the field in the DCB macro instruction. For example, the field reserved for block size would be referred to as DCBBLKSI.

The attributes of each data control block field are defined in the dummy control section. Because each field in the data control block is not necessarily aligned on a fullword boundary, care must be taken when storing or moving data into the field. The length attribute and the alignment of each field can be determined from an assembly listing of the DCBD macro instruction.

The DCBD macro instruction can be coded once to describe all data control blocks, even though their fields differ due to differences in data set organization and access technique. It must not be coded more than once for a single assembly. If it is coded before the end of a control section, it must be followed by a CSECT or DSECT statement to resume the original control section.

Changing an Address in the Data Control Block: The following example illustrates how you can modify a field in the data control block. The DCBD macro instruction defines the symbolic name of each field.

The data set defined by the data control block TEXTDCB is opened for use as both an input and an output data set. When its use as an input data set is completed, the EODAD routine closes the data set temporarily in order to reposition the volume for output. The EODAD routine then uses the dummy control section IHADCB to change the error exit address (SYNAD) from INERROR to OUTERROR.

The EODAD routine loads the address TEXTDCB into register 10, which it uses as a base register for IHADCB. It then moves the address OUTERROR into the DCBSYNAD field of the data control block. This field is a fullword, but contains information in the high order byte which must not be disturbed. For this reason, care is taken to change only the three low order bytes of the field.

```
         OPEN     (TEXTDCB,INOUT)
         ...
EOFEXIT  CLOSE    (TEXTDCB,REREAD),TYPE=T
         LA       10,TEXTDCB
         USING    IHADCB,10
         MVC      DCBSYNAD+1(3),=AL3(OUTERROR)
         B        OUTPUT
INERROR  STM      14,12,SYNADSA+12
         ...
OUTERROR STM      14,12,SYNADSA+12
         ...
TEXTDCB  DCB      DSORG=PS,MACRF=(R,W),DDNAME=TEXTTAPE,
                EODAD=EOFEXIT,SYNAD=INERROR
         DCBD     DSORG=PS
```

SHARING A DATA SET

A data set can be shared by all the tasks of a job step. If requested in the DD statement, a data set can be shared by all the tasks in the system. (Remember that there is only one task in a system with PCP.)

When a data set is shared by several tasks, you must treat it as a serially reusable resource. You must have exclusive control of a data set in order to add or update records, and you must have shared control in order to read records.

In performing a task, you gain exclusive or shared control of a data set by issuing the ENQ and DEQ macro instructions, which are described in "Section I: Supervisor Services." Note that these macro instructions must be used by all of the tasks that process a shared data set.

When you process a direct organization data set, you need to use the ENQ and DEQ macro instructions only when tasks that share a data set do not refer to the same data control block. When all tasks do refer to the same data control block, you must have exclusive control of a block of records that you are updating, but you do not need either shared or exclusive control of the entire data set. You can

request exclusive control of a block of records through the DCB, READ, WRITE, and RELEX macro instructions.

Shared Direct Access Storage Devices: At some installations, a direct access storage device is shared by two or more independent computing systems. Tasks executed on these systems can share data sets stored on the device. For details, refer to the publication IBM System/360 Operating System: System Programmer's Guide.

Part 2: Data Management Processing Procedures

Data Processing Techniques

The operating system allows you to concentrate your efforts on processing the records read or written by the data management routines. Your main responsibility is to describe the data set to be processed, the buffering techniques to be used, and the access method. An access method can be defined as the combination of data set organization and the technique used to process the data. Data access techniques can be divided into two categories -- queued and basic.

QUEUED ACCESS TECHNIQUE

The queued access technique provides GET and PUT macro instructions for transmitting data between main and secondary storage. These macro instructions cause automatic blocking and deblocking of the records stored and retrieved. Anticipatory (look-ahead) buffering and synchronization (overlap) of input and output operations with CPU processing are automatic features of the queued access technique.

Because the operating system controls buffer processing, you can use as many I/O buffers as needed without reissuing GET/PUT macro instructions to fill or empty buffers. Usually, more than one input block is in main storage at any given time to prevent I/O operations from delaying record processing.

Because the operating system synchronizes input/output with processing, you need not test for completion, errors, or exceptional conditions. After a GET or PUT macro instruction is issued, control is not returned to your program until an input area is filled or an output area is available. Exits to error analysis (SYNAD) and end-of-volume or end-of-data (EODAD) routines are automatically taken when necessary.

GET -- Retrieve a Record

The GET macro instruction obtains a record from an input data set. It operates in a logically sequential and device-independent manner. As required, the GET macro instruction schedules the filling of input buffers, deblocks records, and directs input error recovery procedures. For sequential data sets, it will also merge record segments into logical records. After all records have been processed and the GET macro instruction detects an end-of-data indication, the system automatically checks labels on sequential data sets and passes control to your end-of-data (EODAD) routine. If an end-of-volume condition is detected for a sequential data set, the system provides automatic volume switching if the data set extends across several volumes or if concatenated data sets are being processed.

PUT -- Write a Record

The PUT macro instruction places a record into an output data set. Like the GET macro instruction, it operates in a logically sequential and device-independent manner. As required, the PUT macro instruction schedules the emptying of output buffers, blocks records, and handles output error correction procedures. For sequential data sets, it also initiates automatic volume switching and label creation, and also segments records for spanning.

If the PUT macro instruction is directed to a card punch or printer, the system automatically adjusts the number of records or record segments per block of format F or V blocks to 1. Thus, you can specify a record length (LRECL) and block size (BLKSIZE) to provide an optimum block size if the records are temporarily placed on magnetic tape or a direct access volume.

For spanned variable-length records, the block size must be equivalent to one card or one print line. Record size may be greater than block size in this case.

PUTX -- Write an Updated Record

The PUTX macro instruction is used to update a data set or to create an output data set using records from an input data set as a base. PUTX updates, replaces, or inserts records from existing data sets but does not create records or add records from other data sets.

When you use the PUTX macro instruction to update, each record is returned to the data set referred to by a previous GET macro instruction. The buffer containing the updated record is flagged and written back to the same location on the direct access storage device from which it was read. The block is not written out until a GET macro instruction is used for the next buffer, except when a spanned record is to be updated. In that case, the block is written out with the next GET macro instruction.

When the PUTX macro instruction is used to create an output data set, you can add new records by using the PUT macro instruction. As required, the PUTX macro instruction blocks records, schedules the writing of output buffers, and handles output error correction procedures.

BASIC ACCESS TECHNIQUE

The basic access technique provides the READ and WRITE macro instructions for transmitting data between main and secondary storage. This technique is used when the operating system cannot predict the sequence in which the records are to be processed or when you do not want some or all of the automatic functions performed by the queued access technique. Although the system does not provide anticipatory buffering or synchronized scheduling, macro instructions are provided to help you program these functions.

The READ and WRITE macro instructions process blocks, not records. Thus, blocking and deblocking of records is your responsibility. Buffers, allocated either by you or the operating system, are filled or emptied individually each time a READ or WRITE macro instruction is issued. Moreover, the READ and WRITE macro instructions only initiate input/output operations. To ensure that the operation is completed successfully, you must issue a CHECK macro instruction to test the DECB or a WAIT macro instruction and then check the DECB yourself. The number of READ or WRITE macro instructions issued before a CHECK macro instruction is used should not exceed the specified number of channel programs (NCP).

READ -- Read a Block

The READ macro instruction retrieves a data block from an input data set and places it in a designated area of main storage. To allow overlap of the input operation with processing, the system returns control to your program before the read operation is completed. The DECB created for the read operation must be tested for successful completion before processing the record or reusing the DECB.

If an indexed sequential data set is being read, the block is brought into main storage and the address of the desired record is returned to you in the DECB.

When you use the READ macro instruction for BSAM to read a direct data set with spanned records and keys and you specify BFTEK=R in your DCB, the data management routines offset record segments by key length after the first segment of a record. Thus, you can expect the block descriptor word and the segment descriptor word at the same locations in your buffer, or buffers, regardless of whether you read the first segment of a record, which is preceded in the buffer by its key, or you read a subsequent segment, which does not have a key. This facility is called offset reading because the data management routines offset the location of subsequent segments in the buffer by the value of KEYLEN.

You can specify variations of the READ macro instruction according to the organization of the data set being processed and the type of processing to be done by the system as follows:

Sequential
- SF - Read the data set sequentially.
- SB - Reading the data set backward (magnetic tape, format F and U only). When RECFM=FBS, data sets containing a last truncated block cannot be read backwards.

Indexed Sequential
- K - Read the data set.
- KU - read for update. The system maintains the device address of the record; thus, when a WRITE macro instruction returns the record, no index search is required.

Direct
- D - use the direct access method.
- I - locate the block using a block identification.
- K - locate the block using a key.
- F - provide device position feedback.
- X - maintain exclusive control of the block.
- R - provide next-address feedback.
- U - next address can be a capacity record or logical record, whichever occurred first.

WRITE -- Write a Block

The WRITE macro instruction places a data block in an output data set from a designated area of main storage. The WRITE macro instruction can also be used to return an updated record to a data set. To allow overlap of output operations with processing, the system returns control to your program before the write operation is completed. The DECB created for the write operation must be tested for successful completion before the DECB can be reused.

As with the READ macro instruction, you can specify variations of the WRITE macro instruction according to the organization of the data set and the type of processing to be done by the system as follows:

Sequential
- SF - Write the data set sequentially.
- SFR - Write the data set sequentially with next-address feedback.

Indexed Sequential
- K - write a block containing an updated record, or replace a record with an unblocked record having the same key. The record to be replaced need not have been read into main storage.
- KN - write a new record or change the length of a variable-length record.

Direct

SD - write a dummy fixed-length record.
SZ - write a capacity record (R0). The system supplies the data, writes the capacity record, and advances to the next track.
D - use the direct access method.
I - search argument identifies a block.
K - search argument is a key.
A - add a new block.
F - provide record location data (feedback).
X - release exclusive control.

CHECK -- Test Completion of Read/Write Operation

When processing a data set, you can wait and test for completion of a read or write request by issuing a CHECK macro instruction. The system tests for errors and exceptional conditions in the data event control block. Successive CHECK macro instructions issued for the same data set should be issued in the same order as the associated READ/WRITE macro instructions.

The check routine will pass control to the appropriate exit routines specified in the data control block for error analysis (SYNAD) or, for sequential data sets, end-of-data (EODAD). It will also automatically initiate end-of-volume procedures, i.e., volume switching or extending output data sets.

WAIT -- Wait for Completion of a Read/Write Operation

When processing a data set, you can test for completion of any read or write operation by issuing a WAIT macro instruction. The input/output operation will be synchronized with processing, but the DECB will not be checked for errors or exceptional conditions, nor will end-of-volume procedures be initiated. These functions must be tested for and performed by your program.

The WAIT macro instruction can be used to await completion of multiple read/write operations. Each operation must then be checked or tested separately.

Data Event Control Block (DECB)

A data event control block is a 16- to 32-byte area reserved by each READ/WRITE macro instruction. It contains control information and pointers to standard status indicators. It is described in detail in the publication IBM System/360 Operating System: Supervisor and Data Management Macro Instructions.

The DECB is examined by the check routine when the I/O operation is completed to determine if an uncorrectable error or exceptional condition exists. If it does, control is passed to your SYNAD routine. If you have no SYNAD routine, the task is abnormally terminated.

ERROR HANDLING

The basic and queued access techniques both provide special macro instructions for analyzing input/output errors. These macro instructions can be used in SYNAD routines and in error analysis routines that are entered directly when using the basic access technique with indexed sequential data sets.

SYNADAF -- Perform SYNAD Analysis Function

The SYNADAF macro instruction analyzes the status, sense, and exceptional condition code data that is available to your error analysis routine. It produces a descriptive error message that your routine can write into any appropriate data set. The message is in the form of an unblocked variable-length record, but it can be written as a fixed-length record by omitting the block and record length fields that precede the message text.

The text of the message is 120 characters in length, and begins with a field of 36 or 42 blanks; you can use the blank field to add your own remarks to the message. Following is a typical message with the blank field omitted:

```
,TESTJOB ,STEP2   ,283,TA,MASTER  ,READ  ,DATA CHECK     ,0000015,BSAM
```

This message indicates that a data check occurred while reading the fifteenth block of a data set. The data set was identified by a DD statement named MASTER, and was on a magnetic tape volume on unit 283. The name of the job was TESTJOB; the name of the job step was STEP2.

If the error analysis routine is entered because of an input error, the first six bytes of the message (bytes 8-13) contain binary information. If no data was transmitted or if the access method is QISAM, the first six bytes are blank. If the error did not prevent data transmission, the first six bytes contain the address of the input buffer and the number of bytes read. You can use this information to process records from the block; for example, you might print each record after printing the error message. Before printing the message, however, you should replace this binary information with EBCDIC characters.

The SYNADAF macro instruction provides its own save area and makes this area available to your error analysis routine. When used at the entry point of a SYNAD routine, it fulfills the routine's responsibility for providing a save area.

SYNADRLS -- Release SYNADAF Message and Save Areas

The SYNADRLS macro instruction releases the message and save areas provided by the SYNADAF macro instruction. You must issue this macro instruction before returning from the error analysis routine.

ATLAS -- Perform Alternate Track Location Assignment

The ATLAS macro instruction enables your program to recover from permanent input/output errors when processing a data set in direct access storage. After a data check, or in certain missing address marker conditions, you can issue ATLAS to:

- Assign an alternate track to replace the error track.
- Transfer data from the error track to the alternate track.

Use of the ATLAS macro instruction requires a knowledge of channel programming. For this reason, a detailed description of the macro instruction and its use is included in the publication IBM System/360 Operating System: System Programmer's Guide.

If you do not use the ATLAS macro instruction, you can use the IEHATLAS utility program to perform the same function. The principal difference between the macro instruction and the utility program is that the latter provides error recovery only after your own program

has been completed. For a detailed description of IEHATLAS, refer to the publication IBM System/360 Operating System: Utilities.

SELECTING AN ACCESS METHOD

Access methods are identified primarily by the data set organization to which they apply. For instance, we speak of a basic access method for direct organization (BDAM). Nevertheless, there are times when an access method identified with one organization can be used to process a data set usually thought of as organized in a different manner. Thus, a data set is created using the basic access method for sequential organization (BSAM). It is processed using the basic direct access method (BDAM). If the queued access technique is used to process a sequential data set, the access method is referred to as QSAM.

The basic access methods are used for all data organizations, while the queued access methods apply only to sequential and indexed sequential data sets as shown in Table 13.

Table 13. Data Access Methods

Data Set Organization	Access Technique	
	Basic	Queued
Sequential	BSAM	QSAM
Partitioned	BPAM	
Indexed Sequential	BISAM	QISAM
Direct	BDAM	

It is possible to directly control an I/O device while processing any data organization without using a specific access method. The execute channel program (EXCP) macro instruction uses the system functions that provide for scheduling and queuing I/O requests, efficient use of channels and devices, data protection, interruption procedures, error recognition and retry. Complete details about the EXCP macro instruction are in the publication IBM System/360 Operating System: System Programmer's Guide.

OPENING AND CLOSING A DATA SET

Although your program has been assembled, the various data management routines required for I/O operations are not a part of the object code. In other words, your program is not completely assembled until it is initiated for execution. Initiation is accomplished by issuing the OPEN macro instruction. After all data control blocks have been completed, the system ensures that all required access method routines are loaded and ready for use and that all channel command word lists and buffer areas are ready.

Access method routines are selected and loaded according to data control fields that indicate:

- Data organization.
- Buffering technique.
- Access technique.
- I/O unit characteristics.

This information is used by the system to allocate main storage space and load the appropriate routines. These routines, the CCW lists, and buffer areas created automatically by the system remain in main storage until the close routine signals that they are no longer needed by that data control block.

When I/O operations are completed for a data set, a CLOSE macro instruction should be issued to return the data control block to its original status, handle volume disposition, create data set labels, complete writing of queued output buffers, and free main and secondary storage. After the data set has been closed, the data control block can be used for another data set. If you do not close the data set before a task terminates, the operating system closes it automatically. If the data control block is not available to the system at that time, the operating system abnormally terminates the task, and data results can be unpredictable.

An OPEN or CLOSE macro instruction can be used to initiate or terminate processing of more than one data set. Simultaneous opening or closing is faster than issuing separate macro instructions; however, additional storage space is required for each data set specified.

Notes:

1. Two or more data control blocks should never be opened concurrently for output to the same data set on a direct access device. This may result in the end-of-file record written by the CLOSE for one data control block overlaying data associated with another data control block.

2. Two or more data control blocks should never be opened concurrently using the same DDNAME. This is true for both input and output and especially important when using more than one access method. Any action on one DCB that alters the TIOT or JFCB affects the other DCB(s) and thus can cause unpredictable results.

Volume disposition specified in the OPEN or CLOSE macro instruction can be overridden by the system if necessary. However, you need not be concerned; the system automatically requests the mounting and demounting of volumes, depending upon the availability of devices at a particular time.

OPEN -- Initiate Processing of a Data Set

The OPEN macro instruction is used to complete a data control block for an associated data set. The method of processing and the volume positioning instruction in the event of an end-of-volume condition can be specified.

Processing Method: A data set can be processed as either input or output (INPUT, OUTPUT) or a combination of the two (INOUT, OUTIN -- BSAM only). If the data set resides on a direct access volume, records can be updated (UPDAT). Magnetic tape volumes can also be read backwards (RDBACK -- BSAM and QSAM only). If the processing method operand is omitted from the OPEN macro instruction, INPUT is assumed. The operand is ignored by BISAM; it must be specified as OUTPUT when using QISAM to create an indexed sequential data set. You can override the OPEN options INOUT and OUTIN at execution time by using the LABEL parameter of the DD card. Use of this facility is discussed in the publication IBM System/360 Operating System: Job Control Language Reference.

Simultaneous Opening of Data Sets: In this example of the OPEN macro instruction, the data sets associated with three data control blocks are to be opened simultaneously with the indicated options.

```
          OPEN  (TEXTDCB,,CONVDCB,(OUTPUT),PRINTDCB,(OUTPUT))
+         CNOP  0,4
+         BAL   1,*+16 LOAD REG1 W/LIST ADDR.
+         DC    AL1(0) OPTION BYTE
+         DC    AL3(TEXTDCB) DCB ADDRESS
+         DC    AL1(15) OPTION BYTE
+         DC    AL3(CONVDCB) DCB ADDRESS
+         DC    AL1(143) OPTION BYTE
+         DC    AL3(PRINTDCB) DCB ADDRESS
+         SVC   19 ISSUE OPEN SVC
```

Since no processing method operand is specified for TEXTDCB, the system assumes INPUT. Both CONVDCB and PRINTDCB are opened for output. No volume positioning options are specified; thus, the position indicated by the DD statement DISP parameter is used.

At execution time, the SVC 19 instruction passes control to the open routine, which then initiates the three data control blocks and loads the appropriate access method routines.

CLOSE -- Terminate Processing of a Data Set

The CLOSE macro instruction is used to terminate processing of a data set and release it from a data control block. The volume positioning that is to result from closing the data set can also be specified. Volume positioning options are the same as those that can be specified for end-of-volume conditions, as specified in the OPEN macro instruction or the DD statement. An additional volume positioning option, REWIND, is available and can be specified by the CLOSE macro instruction for magnetic tape volumes. REWIND positions the tape at the load point regardless of the direction of processing.

The operating system provides a temporary closing option, CLOSE (TYPE=T), for data sets being processed by BSAM. CLOSE (TYPE = T) causes the RLSE parameter on the DD card to be ignored. When the macro instruction is executed for data sets on magnetic tape or direct access volumes, the system processes labels and repositions the volume as required. However, the data control block maintains its open status. Processing of the data set can be continued at a later stage in your program without reissuing the OPEN macro instruction. Performance is thus improved significantly. Magnetic tape volumes will be repositioned either preceding the first data block or following the last data block of the data set. The presence of tape labels has no effect on repositioning.

Simultaneous Closing of Data Sets: In this example of the CLOSE macro instruction, the data sets associated with three data control blocks are to be closed simultaneously.

```
          CLOSE (TEXTDCB,,CONVDCB,,PRINTDCB)
+         CNOP  0,4
+         BAL   1,*+16 BRANCH AROUND LIST
+         DC    AL1(0) OPTION BYTE
+         DC    AL3(TEXTDCB) DCB ADDRESS
+         DC    AL1(0) OPTION BYTE
+         DC    AL3(CONVDCB) DCB ADDRESS
+         DC    AL1(128)OPTION BYTE
+         DC    AL3(PRINTDCB) DCB ADDRESS
+         SVC   20 ISSUE CLOSE SVC
```

Because no volume positioning operands are specified, the position indicated by the DD statement DISP parameter is used.

At execution time, the SVC 20 instruction passes control to the close routine which terminates processing of the three data sets and returns the three data control blocks to their original status.

End-of-Volume Processing

Control is passed automatically to the data management end-of-volume routine when any of the following conditions is detected:

- End-of-data indicator (input volume).
- Tapemark (input tape volume).
- Filemark (input direct access volume).
- End of reel (output tape volume).
- End of extent (output direct access volume).

You may issue a force end-of-volume (FEOV) macro instruction before the end-of-volume condition is detected.

The end-of-volume routine checks or creates standard trailer labels, if the LABEL parameter of the associated DD statement indicates standard labels. Control is then passed to the appropriate user label routine if it is specified in your exit list.

Multiple volume data sets can be specified in your DD statement whereby automatic volume switching is accomplished by the end-of-volume routine. When an end-of-volume condition exists on an output data set, additional space is allocated as indicated in your DD statement. If no more volumes are specified or if more are required than specified, the storage is obtained from any available volume of the same device type. If no device is available, your job is terminated.

Volume Positioning: When an end-of-volume condition is detected, the system positions the volume according to the disposition specified in the DD statement unless the volume disposition is specified in the OPEN macro instruction. Volume positioning instructions for a sequential data set on tape or direct access can be specified as LEAVE or REREAD.

LEAVE
positions the volume at the logical end of the data set just read or written. If the data set has been read backwards, the logical end is the physical beginning of the data set.

REREAD
positions the volume at the logical beginning of the data set just read or written.

A volume positioning instruction can be specified only if the processing method operand has been specified. It will be ignored if devices other than magnetic tape or direct access are used. It will also be ignored if the number of volumes exceeds the number of available units.

For magnetic tape volumes, positioning varies according to the direction of the last input operation and the existence of tape labels. If the tape was last read forward:

LEAVE
will position a labeled tape following the tapemark that follows

the data set trailer label group; an unlabeled volume following the tapemark that follows the last block of the data set.

REREAD
will position a labeled tape preceding the data set header label group; an unlabeled tape preceding the first block of the data set.

If the tape was last read backwards:

LEAVE
will position a labeled tape preceding the data set header label group; an unlabeled tape preceding the first block of the data set.

REREAD
will position a labeled tape following the tape mark that follows the data set trailer label group; an unlabeled tape following the tape mark that follows the last block of the data set.

FEOV -- Force End of Volume

The FEOV macro instruction directs the operating system to initiate end-of-volume processing before the physical end of the current volume is reached. If another volume has been specified for the data set, volume switching takes place automatically. The volume positioning options REWIND and LEAVE are available.

The FEOV macro instruction can only be used when processing data sequentially, i.e., BSAM and QSAM.

Buffer Acquisition and Control

The buffering facilities of the operating system provide several methods of acquisition and control. Each buffer, i.e., main storage area used for intermediate storage of input/output data, usually corresponds in length to the size of a block in the data set being processed. When using the queued access technique, any reference to a buffer actually refers to the next record, i.e., buffer segment.

You can assign more than one buffer to a data set by associating the buffer with a buffer pool. A buffer pool must be constructed in a main storage area allocated for a given number of buffers of a given length.

Buffer segments and buffers within the buffer pool are controlled automatically by the system when the queued access technique is used. However, you can terminate processing of a buffer by issuing a release (RELSE) macro instruction for input or a truncate (TRUNC) macro instruction for output. Two buffering techniques, simple and exchange, can be used to process a sequential data set. Only simple buffering can be used to process an indexed sequential data set.

If you use the basic access technique, you can use buffers as work areas rather than as intermediate storage areas. They can be controlled either directly by using the GETBUF/FREEBUF macro instruction, or dynamically by requesting dynamic buffering in your DCB macro instruction and your READ/WRITE macro instruction. If you request dynamic buffering, the system will automatically provide a buffer each time a READ macro instruction is issued. That buffer will be freed when you issue a WRITE or FREEDBUF macro instruction.

BUFFER POOL CONSTRUCTION

Buffer pool construction can be accomplished in any of three ways:

- Statically using the BUILD macro instruction.
- Explicitly using the GETPOOL macro instruction.
- Automatically by the system when the data set is opened.

If QSAM simple buffering is used, the buffers are automatically returned to the pool when the data set is closed. If the buffer pool is constructed explicitly or automatically, the main storage area must be returned to the system by using the FREEPOOL macro instruction.

In many applications, singleword or doubleword alignment of a block within a buffer is important. You can specify in the data control block that buffers are to start on either a doubleword or a fullword boundary that is not also a doubleword boundary (BFALN=D or F). If doubleword alignment is specified for format V records, the fifth byte of the first record in the block is so aligned. For that reason, fullword alignment must be requested to align the first byte of the variable-length record on a doubleword boundary. The alignment of the records following the first in the block depends on the length of the previous record.

If the BUILD macro instruction is used to construct the buffer pool, alignment depends on the alignment of the first byte of the reserved storage area.

When you process multiple QISAM data sets, you can use a common buffer pool. To do this, however, you must use the BUILD macro instruction to reformat the buffer pool before opening each data set.

BUILD -- Construct a Buffer Pool

When you know, prior to program assembly, both the number and size of the buffers required for a given data set, you can reserve an area of appropriate size to be used as a buffer pool. Any type of area can be used -- a predefined storage area, or an area of coding no longer needed, for example.

A BUILD macro instruction, issued during execution of your program, structures the reserved storage area into a buffer pool. The address of the buffer pool must be the same as that specified for the buffer pool control block (BUFCB) in your data control block. The buffer pool control block is an 8-byte field preceding the buffers in the buffer pool. The number (BUFNO) and length (BUFL) of the buffers must also be specified.

When the data set using the buffer pool is closed, you can reuse the area as required. You can also reissue the BUILD macro instruction to reconstruct the area into a new buffer pool to be used by another data set.

You can assign the buffer pool to two or more data sets that require buffers of the same length. To do this, you must construct an area large enough to accommodate the total number of buffers required at any one time during execution. That is, if each of two data sets requires five buffers (BUFNO=5), the BUILD macro instruction should specify ten buffers. The area must also be large enough to contain the 8-byte buffer pool control block.

BUILDRCD -- Build a Buffer Pool and a Record Area

The BUILDRCD macro instruction performs the same functions as the BUILD macro instruction and the following functions, as well:

- It provides the logical record interface necessary for a sequential data set accessed by QSAM in the locate mode and having a record format of VS or VBS. Logical record interface, unlike segment interface, enables the user to access an entire logical record, not just a segment.

- It links a record area to the buffer control block by extending the buffer control block to twelve bytes. Thus, a spanned record can be assembled or segmented in the record area.

GETPOOL -- Get a Buffer Pool

If a specified area is not reserved for use as a buffer pool, or you want to defer specifying the number and length of the buffers until execution of your program, you should use the GETPOOL macro instruction. This facility enables you to vary the size and number of buffers according to the needs of the data set being processed.

The GETPOOL macro instruction structures a main storage area allocated by the system into a buffer pool, assigns a buffer pool control block, and associates the pool with a specific data set. The GETPOOL macro instruction should be issued either before opening the data set or during your DCB exit routine.

Automatic Buffer Pool Construction

If you have requested a buffer pool and have not used an appropriate macro instruction by the end of your DCB exit routine, the system automatically allocates main storage space for a buffer pool. The buffer pool control block is also assigned and the pool is associated with a specific data set. If you are using the basic access technique to process an indexed sequential or direct data set, you must indicate dynamic buffer control. Otherwise, the system does not construct the buffer pool automatically.

FREEPOOL -- Free a Buffer Pool

Any buffer pool assigned to a data set either automatically by the OPEN macro instruction (except when dynamic buffer control is used) or explicitly by the GETPOOL macro instruction must be released before your program is terminated. The FREEPOOL macro instruction should be issued to release the main storage area as soon as the buffers are no longer needed. As a general rule, when you are using the queued access technique, an output data set should be closed first to ensure that all the records have been written out. However, when using exchange buffering or when processing an indexed sequential data set using the queued access technique, the buffer pool must not be released until all the data sets have been closed.

Constructing a Buffer Pool: The following examples illustrate several possible methods of constructing a buffer pool. The examples do not consider the method of processing or controlling the buffers in the pool.

```
          ...                                   Processing
          BUILD     INPOOL,10,52                Structure a buffer pool
          OPEN      (INDCB,,OUTDCB,(OUTPUT))
          ...                                   Processing
ENDJOB    CLOSE     (INDCB,,OUTDCB)
          ...                                   Processing
          RETURN                                Return to System Control
INDCB     DCB       BUFNO=5,BUFCB=INPOOL,EODAD=ENDJOB,---
OUTDCB    DCB       BUFNO=5,BUFCB=INPOOL,---
          CNOP      0,8                         Force boundary alignment
INPOOL    DS        CL564                       Buffer pool
```

In the first example, a static storage area named INPOOL is allocated during program assembly. The BUILD macro instruction, issued during execution, arranges the buffer pool into ten buffers, each 52 bytes long. Five buffers are assigned to INDCB and five to OUTDCB, as specified in the DCB macro instruction for each. The two data sets share the buffer pool because both specify INPOOL as the buffer pool control block. Notice that an additional eight bytes have been allocated for the buffer pool to contain the buffer pool control block.

```
          ...
          GETPOOL   INDCB,10,52          Construct a 10-buffer pool
          GETPOOL   OUTDCB,5,112         Construct a  5-buffer pool
          OPEN      (INDCB,,OUTDCB,(OUTPUT))
          ...
ENDJOB    CLOSE     (INDCB,,OUTDCB)
          FREEPOOL  INDCB                Release buffer pools after all
          FREEPOOL  OUTDCB                I/O is complete
          ...
          RETURN                         Return to System Control
INDCB     DCB       DSORG=PS,BFALN=F,LRECL=52,RECFM=F,EODAD=ENDJOB,---
OUTDCB    DCB       DSORG=IS,BFALN=D,LRECL=52,KEYLEN=10,BLKSIZE=104,    C
                RKP=0,RECFM=FB,---
```

In the second example, two buffer pools are constructed explicitly by the GETPOOL macro instructions. Ten input buffers are provided, each 52 bytes long, to contain one fixed-length record; five output buffers are provided, each 112 bytes long, to contain two blocked records plus an 8-byte count field (required by the Indexed Sequential Access Method). Notice that both data sets are closed before the buffer pools are released by the FREEPOOL macro instructions. The same procedure should be used if the buffer pools were constructed automatically by the OPEN macro instruction.

BUFFER CONTROL

There are four techniques that can be used to control the buffers used by your program. The advantages of each depend to a great extent upon the type of job you are doing. Both simple and exchange buffering are provided for the queued access technique. The basic access technique provides for either direct or dynamic buffer control.

Although only simple buffering can be used to process an indexed sequential data set, buffer segments and buffers within a buffer pool are controlled automatically by the operating system.

In addition, the queued access technique provides four processing modes that determine the extent of data movement in main storage. Move, data, locate, or substitute mode processing can be specified for either the GET or PUT macro instructions. The buffer processing mode

is specified in the MACRF field of the DCB macro instruction. The movement of a record is determined as follows:

- Move mode: The record is moved from an input buffer to your work area, or from your work area to an output buffer.

- Data mode (QSAM V format spanned records only): The same as the move mode except only the data portion of the record is moved.

- Locate mode: The record is not moved. Instead, the address of the next input or output buffer is placed in register 1.

  For QSAM format V spanned records, the record is not moved. Instead, if logical record interface has been requested by specifying BFTEK=A or by issuing the BUILDRCD macro instruction, the address returned in register 1 points to a record area where the spanned record is assembled or segmented.

- Substitute mode: The record is not moved. Instead, the address of the next input or output buffer is interchanged with the address of your work area.

Two processing modes of the PUTX macro instruction can be used in conjunction with a GET-locate macro instruction. The update mode returns an updated record to the data set from which it was read; the output mode transfers an updated record to an output data set. There is no actual movement of data in main storage. The processing mode must be specified in the MACRF parameter of the DCB macro instruction.

If you use the basic access technique, you can control buffers in one of two ways:

- Directly using the GETBUF macro instruction to retrieve a buffer constructed as described above. A buffer can then be returned to the pool using the FREEBUF macro instruction.

- Dynamically by requesting a dynamic buffer in your READ/WRITE macro instruction. This technique can be used when processing an indexed sequential or direct organization data set. If you request dynamic buffering, the system will automatically provide a buffer each time a READ macro instruction is issued. The buffer is supplied from a buffer pool which is created by the system when the data control block of the data set is opened. The buffer will be released (returned to the pool) upon completion of a WRITE macro instruction when you are updating. If you do not update the record in the buffer and thus release the buffer when the record is written, the FREEDBUF macro instruction may be used. If you are processing an indexed sequential data set, the buffer is automatically released upon the next issuance of the READ macro instruction if there has been no intervening WRITE or FREEDBUF macro instruction issued.

Simple Buffering

The term "simple" buffering refers to the relationship of segments within the buffer. All segments in a simple buffer are contiguous in main storage and are always associated with the same data set. When the buffer pool is constructed, the system creates a channel command word (CCW) for each buffer in the buffer pool. For this reason, each record must be physically moved from an input buffer segment to an output buffer segment. It can be processed within either segment or in a work area.

If you use simple buffering, records of any format can be processed. New records can be inserted and old records deleted as required to create a new data set. Records can be moved and processed as follows:

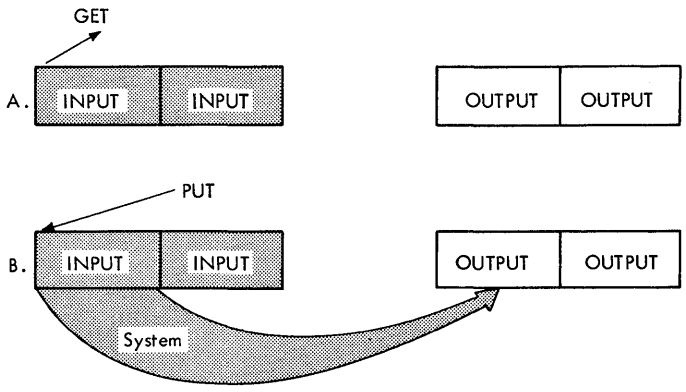
- Processed in an input buffer and then moved to an output buffer (GET-locate, PUT-move/PUTX-output).
- Moved from an input buffer to an output buffer where it can be processed (GET-move, PUT-locate).
- Moved from an input buffer to a work area where it can be processed and then moved to an output buffer (GET-move, PUT-move).
- Processed in an input buffer and returned to the data set (GET-locate, PUTX-update).

The following examples illustrate the control of simple buffers and the processing modes that can be used. The buffer pools may have been constructed in any way previously described.

Simple Buffering -- GET-locate, PUT-move/PUTX-output: The GET macro instruction (step A, Figure 23) locates the next input record to be processed. Its address is returned in register 1 by the system. The address is passed to the PUT macro instruction in register 0.

The PUT macro instruction (step B, Figure 23) specifies the address of the record in register 0. The system then moves the record to the next output buffer.

Note: The PUTX-output macro instruction can be used in place of the PUT-move macro instruction. However, processing will be as described under exchange buffering (see PUT-substitute).




```
NEXTREC  GET INDCB
         ...
         LR  0,1
         PUT OUTDCB,(0)
         B   NEXTREC
INDCB    DCB MACRF=(GL),---
OUTDCB   DCB MACRF=(PM),---
```

Figure 23. Simple Buffering (GL,PM)

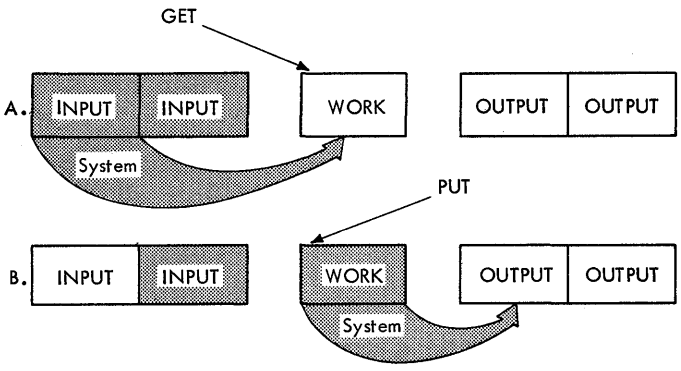
Simple Buffering -- GET-move, PUT-locate: The PUT macro instruction locates the address of the next available output buffer. Its address is returned in register 1 and is passed to the GET macro instruction in register 0.

The GET macro instruction specifies the address of the output buffer into which the system moves the next input record.

A filled output buffer is not written until the next PUT macro instruction is issued.

Simple Buffering -- GET-move, PUT-move: The GET macro instruction (step A, Figure 24) specifies the address of a work area into which the system moves the next record from the input buffer.

The PUT macro instruction (step B, Figure 24) specifies the address of a work area from which the system moves the record into the next output buffer.




```
NEXTREC   GET INDCB,WORKAREA
          ...
          PUT OUTDCB,WORKAREA
          B   NEXTREC
WORKAREA  DS  CL50
INDCB     DCB MACRF=(GM),---
OUTDCB    DCB MACRF=(PM),---
```

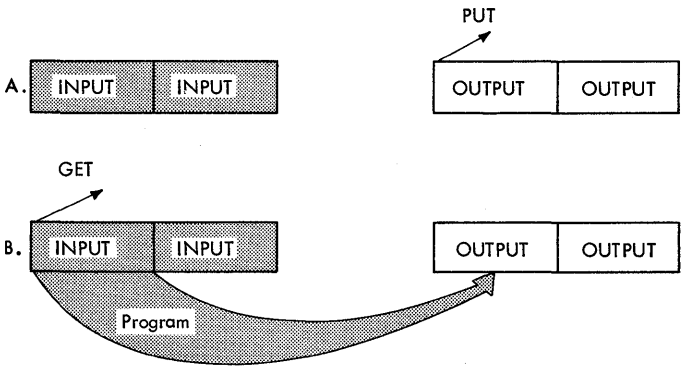
Figure 24. Simple Buffering (GM, PM)

Simple Buffering -- GET-locate, PUT-locate: The PUT macro instruction (step A, Figure 25) locates the address of the next available output buffer. The address is returned in register 1.

The GET macro instruction (step B, Figure 25) locates the address of the next input buffer. Its address is returned in register 1. You must then move the record from the input buffer to the output buffer. Processing can be done either before or after the move operation.

A filled output buffer is not written until the next PUT macro instruction is issued.

Note: If records other than format F are being moved, the length attribute of the MVC instruction must be changed as shown. If the record is more than 256 bytes, you will have to code a move routine to process the complete record.




```
NEXTREC   GET INDCB
          LR  7,1
          PUT OUTDCB
          LR  6,1
See Note  USING  IHADCB,5
          LA  5,INDCB
          LH  4,DCBLRECL
          SH  4,=H'1'
          EX  4,MOVEREC
          ...
          B   NEXTREC
MOVEREC   MVC 0(1,6),0(7)
INDCB     DCB MACRF=(GL),
              EODAD=EOF,---
OUTDCB    DCB MACRF=(PL),---
          DCBD DSORG=(LR)
EOF       ...
```

● Figure 25. Simple Buffering (GL, PL)

Exchange Buffering

The term "exchange" buffering refers to the relationship of segments within a buffer. All the segments in an exchange buffer are not necessarily contiguous in main storage, nor are they always associated with the same data set. When the buffer pool is constructed, the system creates a channel command word (CCW) for each buffer segment in the buffer. This facility makes it possible to "exchange" the CCWs of different storage locations.

To use exchange buffering, you must provide a work area comparable in size and alignment to a buffer segment. That work area is substituted for the next buffer segment. That is, the storage areas change roles. The CCW created for the buffer segment actually points to the work area.

Why use exchange buffering? Because there is no need to move the record. This means a considerable savings in processing time. On the other hand, exchange buffering is of no advantage unless substitute mode or PUTX-output mode is used.

The implementation of exchange buffering during execution of your program depends on a number of factors:

- Input and output buffers must be of the same size and alignment.
- Records must be blocked format F or unblocked.
- Track overflow cannot be used with blocked format F records.
- GET-move and PUT-locate modes cannot be used.
- Unit record devices must not be specified.

If you request exchange buffering, but it cannot be implemented, the system automatically provides simple buffering. Move mode processing is used in place of substitute mode.

After opening the data set, you can test the DCBCIND1 field of the data control block to determine if simple buffering was substituted for exchange buffering because of inconsistencies in the data control block information. The eighth bit of the DCBCIND1 field is 1 for exchange buffering and 0 for simple buffering.

If your records are blocked format F, each segment is aligned as specified in the DCBBFALN field. Therefore, your buffer length (DCBBUFL) must be large enough to contain segments that are a multiple of 16 bytes. Otherwise, the specified boundary alignment cannot be acnieved; simple buffering is used and only the first byte in the first record is aligned as specified.

To reopen a DCB that has been opened for exchange buffering, you must first do the following:

- Close all DCBs using the buffer pool associated with the DCB to be reopened.
- Issue a FREEPOOL macro instruction specifying the DCB to be reopened.

There are two possible error conditions that cannot be prechecked by the system:

- Word alignment that does not correspond to the characteristics of the machine. If, for example, you expect to process your data on a model 65 or 75, your record length should be a multiple of 16; on a model 50, a multiple of 8; on a model 40, a multiple of 4. No error will result if the records are processed on a smaller system.

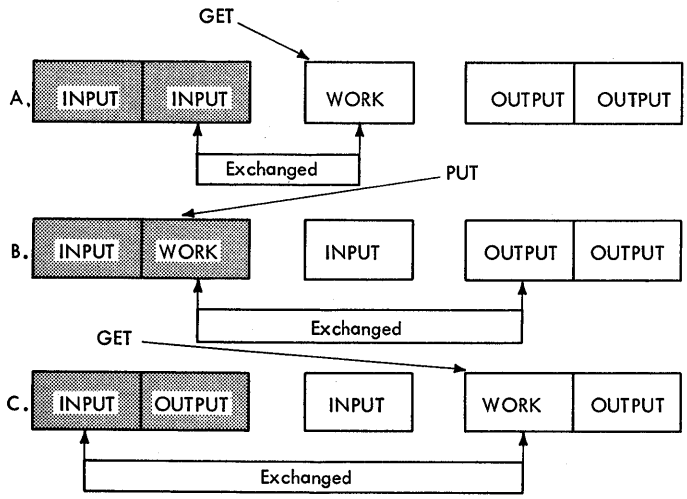
- An I/O device that transfers the data faster than the CPU can exchange the addresses in the CCW.

The following examples illustrate the control of exchange buffers and the corresponding processing modes that can be used. The buffer pools may have been constructed in any way previously described.

Exchange Buffering -- GET-substitute, PUT-substitute: The GET macro instruction (step A, Figure 26) specifies the address of a work area. The work area address is exchanged for the address of the next input record returned in register 1. After processing, the address of the record is passed to the PUT macro instruction.

The PUT macro instruction (step B, Figure 26) specifies the address of the output record. The output record address is exchanged for the address of the next output buffer available for use as a work area. The work area address, returned in register 1, is passed to the GET macro instruction (step C, Figure 26) in register 0.

Notice that as the areas are exchanged there is no movement of data. Output records are contained in the original input area and vice versa, but are logically associated with the correct data set.

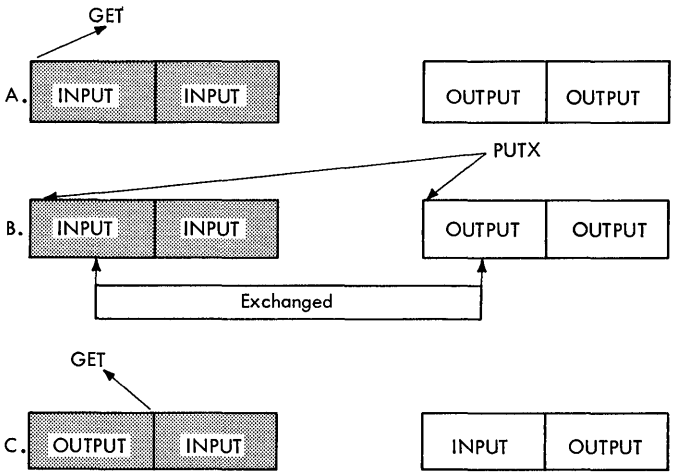



```
            LA   0,WORKAREA
NEXTREC     GET  INDCB,(0)
            LR   0,1
            ...
            PUT  OUTDCB,(0)
            LR   0,1
            B    NEXTREC
WORKAREA    DS   CL50
INDCB       DCB  MACRF=(GT),---
OUTDCB      DCB  MACRF=(PT),---
```

Figure 26. Exchange Buffering (GT, PT)

Exchange Buffering -- GET-locate, PUTX-output: The GET macro instruction (step A, Figure 27) locates the address of the next input record. The address is returned in register 1. The record must be processed in the buffer segment before the PUTX macro instruction (step B, Figure 27) is issued. The PUTX macro instruction specifies the address of both the input and output data control block. The two buffer segments are exchanged without any movement of data. The GET macro instruction (step C, Figure 27) locates the next record to be processed.

Notice that the DCB macro instruction for the output data set specifies move mode; this is required.




```
NEXTREC  GET INDCB
         ...
         PUTX OUTDCB,INDCB
         B   NEXTREC
INDCB    DCB MACRF=(GL),---
OUTDCB   DCB MACRF=(PM),---
```

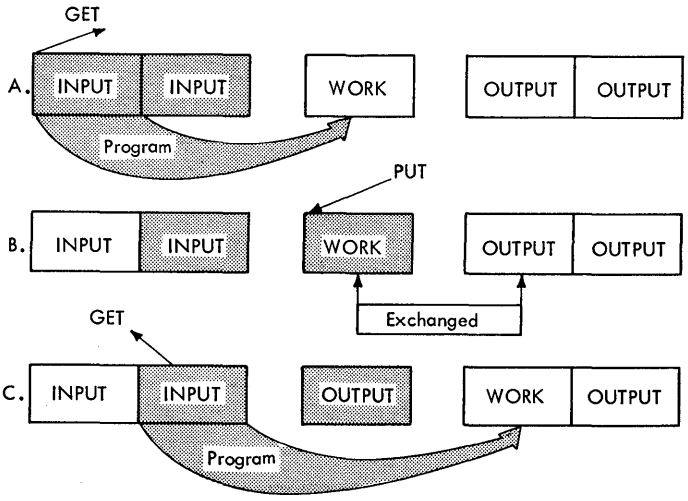
Figure 27. Exchange Buffering (GL, PM)

Exchange Buffering -- GET-locate, PUT-substitute: The GET macro instruction (step A, Figure 28) locates the next input record. Its address is returned in register 1. You must then move the record to a work area. Processing can be done either before or after the move operation.

The PUT macro instruction (step B, Figure 28) specifies the address of the work area containing the next output record. The system returns the address of the next output buffer available for use as a work area in register 1. That address is passed to the move (MVC) instruction in register 6.

The GET macro instruction (step C, Figure 28) locates the next input record. You must then move the record to the new work area. Notice that the previous work area has become a part of the output buffer (step C).

Note: If records other than format F are being moved, the length attribute of the MVC instruction must be changed as shown. If the record is more than 256 bytes long, you must code a move routine to process the complete record.




```
                 LA    6,WORKAREA
NEXTREC          GET   INDCB
                 LR    7,1
See Note         USING IHADCB,5
                 LA    5,INDCB
                 LH    4,DCBLRECL
                 SH    4,=H'1'
                 EX    4,MOVEREC
                 ...
                 PUT   OUTDCB,(6)
                 LR    6,1
                 B     NEXTREC
MOVEREC          MVC   0(1,6),0(7)
WORKAREA         DS    CL50
INDCB            DCB   MACRF=(GL),---
OUTDCB           DCB   MACRF=(PT),---
                 DCBD  DSORG=(LR)
```

Figure 28. Exchange Buffering (GL, PT)

Buffering Techniques and GET/PUT Processing Modes: As you can see from the previous examples, the most efficient coding is achieved by using automatic buffer pool construction, and GET-locate and PUTX-output with either simple or exchange buffering. Table 14 summarizes the combinations of buffering techniques and processing modes that can be used. Notice, for example, that if you use PUT-locate and GET-substitute, you must provide a work area and you must also move the record from the work area to the output buffer.

• Table 14. Buffering Technique and GET/PUT Processing Modes

Output Buffering: →	Simple		Exchange		Simple		Exchange		Simple		Simple		Exchange		Simple		Exchange	
Input Buffering: → Simple Actions ↓	GET-move, PUT-locate	GET-move, PUT-move	GET-move, PUT-move	GET-move, PUT-substitute	GET-locate, PUT-locate	GET-locate, PUT-move	GET-locate, PUT-move	GET-locate, PUT-substitute	GET-locate (logical record), PUT-locate	Input Buffering: Exchange	GET-locate, PUT-locate	GET-locate, PUT-move	GET-locate, PUT-move	GET-locate, PUT-substitute	GET-substitute, PUT-locate	GET-substitute, PUT-move	GET-substitute, PUT-move	GET-substitute, PUT-substitute
Program must move record					X			X	X		X			X	X			
System moves record	X	X	X	X		X	X					X	X			X	X	
System moves record segment									X									
Record is not moved																		X
Work Area required		X	X	X				X						X	X	X	X	X
PUTX - output can be used						X	X					X	X					

RELSE -- Release an Input Buffer

When using the queued access technique to process a sequential or indexed sequential data set, you can direct the system to ignore the remaining records in the input buffer being processed. The next GET macro instruction retrieves a record from another buffer. If format V spanned records are being used, the next logical record obtained may begin on any segment in any subsequent block.

If you are using move mode, the buffer is made available for refilling as soon as the RELSE macro instruction is issued. When used with locate mode, the system does not refill the buffer until the next GET macro instruction is issued. If a PUTX macro instruction has been used, the block is written before the buffer is refilled.

TRUNC -- Truncate an Output Buffer

When using the queued access technique to process a sequential data set, you can direct the system to write a short block. The first record in the next buffer is the next record processed by a PUT/PUTX-output mode.

If the locate mode is being used, the system assumes that a record has been placed in the buffer segment pointed to by the last PUT macro instruction.

The last block of a data set is truncated by the close routine.

Note: A data set containing format F records with truncated blocks cannot be read from direct access storage as efficiently as standard format F data sets.

GETBUF -- Get a Buffer From a Pool

The GETBUF macro instruction can be used with the basic access technique to request a buffer from a buffer pool constructed by the BUILD, GETPOOL, or OPEN macro instruction. The address of the buffer is returned by the system in a register specified by you when the macro instruction is issued. If no buffer is available, the register contains zeros instead of an address.

FREEBUF -- Return a Buffer to a Pool

The FREEBUF macro instruction is used with the basic access technique to return a buffer to the buffer pool from which it was obtained by a GETBUF macro instruction. Although the buffers need not be returned in the order in which they were obtained, they must be returned when they are no longer needed.

FREEDBUF -- Return a Dynamic Buffer to a Pool

Any buffer obtained using the dynamic buffer option must be released before it can be used again. When you are processing a direct data set, if you do not update the block in the buffer and thus free the buffer when the block is written, you must use the FREEDBUF macro instruction. If there is an uncorrectable input/output error, the control program releases the buffer. When you are processing an indexed sequential data set, if you do not update the block in the buffer or if there is an uncorrectable input error, the control program releases the buffer when the next READ macro instruction is issued on the same DECE, or you may use the FREEDBUF macro instruction.

To effect the release, you must specify the address of the DECB that was created when the block was read using the dynamic buffering option, as well as the address of the data control block associated with the data set being processed.

Processing a Sequential Data Set

Data sets residing on all volumes other than direct access must be processed sequentially. In addition, a data set residing on a direct access volume, regardless of organization, can be processed sequentially. This feature of the operating system allows you to write your program with little regard for the type of device to be used when the program is executed. Naturally, there are restrictions against the use of certain device-dependent macro instructions and processing options.

Either the queued or basic access technique may be used to store and retrieve the records of a sequential data set. Additionally, a technique called chained scheduling can be used to accelerate the input/output operations required for a sequential data set.

DATA FORMAT -- DEVICE TYPE CONSIDERATIONS

Both the record format (RECFM) and device-dependent (DEVD) information must be provided to the operating system prior to execution of your program. This information can be supplied by a DCB macro instruction, a DD statement, or a data set label. The DCB subparameters for the DD statement differ slightly from those described here. A complete description of the DD statement and a glossary of DCB subparameters is contained in the publication IBM System/360 Operating System: Job Control Language.

The record format (RECFM) parameter of the DCB macro instruction specifies the characteristics of the records in the data set as fixed length (F), variable-length (V), or undefined length (U). Fixed-length, blocked records (FB) can be specified as standard (FBS), i.e., there are no truncated (short) blocks or unfilled tracks within the data set, with the possible exception of the last block or track. If the data set resides on a direct access volume, the track overflow feature (T) cannot be specified for the basic access technique.

If you plan to read a data set backwards or to extend it at a later time, proceed as follows when coding RECFM for the creation of a sequential data set:

- If you know you will not have any truncated blocks, you can specify RECFM=FBS.
- If you are uncertain about whether you will have a truncated block, you should specify RECFM=FB.

As an optional feature, a control character can be contained in each record. This control character will be recognized and processed if the data set is printed or punched. The control characters are transmitted on both tapes and direct access devices. The presence of a control character is indicated by M or A in the RECFM field of the data control block. M denotes machine code; A denotes American National Standard Code for Information Interchange (ASCII). If either M or A is specified, the character must be present in every record; the printer space (PRTSP) or stacker select (STACK) field of the data control block is ignored. The optional control character must be in the first byte of format F or U records and in the fifth byte of format V records. Control character codes are listed in Appendix E.

The device-dependent (DEVD) parameter of the DCB macro instruction specifies the type of device on which the data set's volume resides:

TA - magnetic tape
PT - paper tape reader
PR - printer
PC - card punch
RD - card reader
DA - direct access

MAGNETIC TAPE (TA)

Format F, V, or U records are acceptable for magnetic tape. However, format V records are not acceptable on 7-track tape if the

data conversion feature[1] is not available. Data blocks should be at least 18 bytes long. If a data check occurs when you are reading a data block shorter than 18 bytes, the error recovery procedures skip the data block. When you create a tape data set with variable-length record format, the control program pads any data block shorter than 18 bytes. It pads to the right with binary zeroes so that the data block length equals 18 or block size, whichever is shorter.

Tape density (DEN) specifies the recording density in bits per inch per track, as shown in Table 15. If this information is not supplied, the highest applicable density is assumed.

• Table 15. Tape Density (DEN) Values

DEN Value	Recording Density Model 2400			
	7-Track	9-Track	9-Track (phase encoded)	9-Track (dual density)
0	200	-	-	-
1	556	-	-	-
2	800	800	-	800[1]
3	-	-	1600	1600[2]

[1]Non-return-to-zero IBM (NRZI) mode
[2]Phase encoding (PE) mode

The track recording technique (TRTCH) for 7-track tape can be specified as:

C - data conversion is to be used.
E - even parity is to be used; if omitted, odd parity is assumed.
T - BCDIC to EBCDIC translation is required.

PAPER TAPE READER (PT)

The paper tape reader accepts format F or U records. Each format U record is followed by an end-of-record character. Data read from paper tape is optionally converted into the System/360 internal representation of one of six standard paper tape codes. Any character found to have a parity error will not be converted when the record is transferred into the input area. Characters deleted in the conversion process are not counted in determining the block size.

The following symbols indicate the code in which the data was punched. If this information is omitted, I is assumed.

I - IBM BCD perforated tape and transmission code (8 tracks).
F - Friden (8 tracks).
B - Burroughs (7 tracks).
C - National Cash Register (8 tracks).
A - ASCII (8 tracks).
T - Teletype (5 tracks).
N - No conversion.

---

[1]Data conversion makes it possible to write eight binary bits of data on seven tracks. Otherwise, only six bits of an 8-bit byte are recorded. The length field of format V records contains binary data and is not recorded correctly without data conversion.

Note: When using QSAM, the processing mode must be move mode.

CARD READER AND PUNCH (RD/PC)

Format F, V, or U records are acceptable to both the reader and punch. The device control character, if specified in the RECFM parameter, is used to select the stacker; it is not punched. The first four bytes of format V records or record segments (record or segment descriptor word) are not punched.

Each punched card corresponds to one physical record. Therefore, you should restrict the maximum record size to 80 (EBCDIC mode) and 160 (column binary mode) data bytes. If mode (C) is used, the DCB parameters BLKSIZE, LRECL, and BUFL must be specified as 160. You can specify the read/punch mode of operation (MODE) as either card image (column binary) mode (C) or EBCDIC mode (E). If this information is omitted, E is assumed.

Stacker selection (STACK) can be specified as either 1 or 2 to indicate which bin is to receive the card. If it is not specified, 1 is assumed.

Note: When QSAM is used, punch error correction on the IBM 2540 Card Read Punch is automatically performed only for data sets using three or more buffers without the chained scheduling feature.

PRINTER (PR)

Records of format F, V, or U are acceptable to the printer. The first four bytes (record descriptor word) of format V records are not printed. The carriage control character, if specified in the RECFM parameter, is not printed. However, the system does not position the printer to channel 1 for the first record.

Because each line of print corresponds to one record, the record length should not exceed the length of one line on the printer. For variable-length spanned records, each line corresponds to one record segment, and block size should not exceed the length of one line on the printer.

If carriage control characters are not specified, you can indicate printer spacing (PRTSP) as 0, 1, 2, or 3. If it is not specified, 1 is assumed.

DIRECT ACCESS (DA)

Direct access devices accept records of format F, V, or U. If the records are to be read or written with keys, the key length (KEYLEN) must be specified. In addition, the operating system has a standard track format for all direct access volumes. Each track contains data information as well as certain "nondata" or control information such as:

- The address of the track.
- The address of each record.
- The length of each record.
- Gaps between areas.

A complete description of track format is contained in the section "Direct Access Volume Characteristics." Your only concern in creating a sequential data set is to allow for an 8-byte track descriptor record (capacity record or R0) when requesting space on a direct

access volume. In addition, "device overhead," which varies with the device, must be allocated for each block on the track.

SEQUENTIAL DATA SETS -- DEVICE CONTROL

The operating system provides you with six macro instructions for controlling input/output devices. Each is, to varying degrees, device-dependent. Therefore, you must exercise some care if you wish to achieve device independence.

When using the queued access technique, only unit record equipment can be controlled directly. When using the basic access technique, limited device independence can be achieved between magnetic tape and direct access devices. All read or write operations must be checked before issuing a device control macro instruction.

CNTRL -- Control an I/O Device

The CNTRL macro instruction provides a number of device-dependent control functions:

- Card reader stacker selection (SS).
- Printer line spacing (SP).
- Printer carriage control (SK).
- Magnetic tape backspace (BSR) over a specified number of blocks.
- Magnetic tape backspace (BSM) past a tapemark and forward space over the tapemark.
- Magnetic tape forward space (FSR) over a specified number of blocks.
- Magnetic tape forward space (FSM) past a tapemark and a backspace over the tapemark.

Backspacing moves the tape toward the load point; forward spacing moves the tape away from the load point.

Note: The CNTRL macro instruction cannot be used with an input data set containing variable-length records on the card reader.

PRTOV -- Test for Printer Overflow

The PRTOV macro instruction tests for channel 9 or 12 of the printer carriage control tape. An overflow condition will cause either an automatic skip to channel 1 or, if specified, transfer of control to your routine for overflow processing.

If the data set specified in the data control block is not a printer, no action is taken.

SETPRT -- Load Character Set for UCS Printer

The SETPRT macro instruction indicates the character set to be used by a 1403 printer with the Universal Character Set feature. It thus allows your program to change character sets during execution; as an option, it allows lower-case alphabetic characters to be printed in uppercase when no uppercase/lowercase print chain is available.

When issued, the SETPRT macro instruction loads a special UCS buffer from the system library. The library contains images of

standard IBM character sets and of special user-designed character sets. The operator can be requested to verify the loaded image after mounting the appropriate print chain or train.

The SETPRT macro instruction can be used to block or unblock printer data checks. When data checks are blocked, unprintable characters are treated as blanks and do not cause an error condition.

BSP -- Backspace a Magnetic Tape or Direct Access Volume

The BSP macro instruction backspaces one block on the magnetic tape or direct access volume being processed. The block can then be reread or rewritten. An attempt to rewrite the block destroys the contents on the remainder of the tape or track.

The direction of movement is toward the load point or beginning of allocated area. You may not use the BSP macro instruction if the track overflow option was specified or if the CNTRL, NOTE, or POINT macro instructions are used. The BSP macro instruction should be used only when other device control macro instructions could not be used for backspacing.

NOTE -- Return the Relative Address of a Block

The NOTE macro instruction requests the relative address of the block just read or written. The feedback identifies the block for subsequent repositioning of the volume.

The feedback provided by the operating system is returned in general register 1. The address is in the form of a 4-byte relative block address for magnetic tape; for a direct-address device, it is a 4-byte relative track address and the amount of unused space available on the track.

POINT -- Position to a Block

The POINT macro instruction causes repositioning of a magnetic tape or direct access volume to a specified block in the data set. The next read or write operation begins at this block.

SEQUENTIAL DATA SETS -- DEVICE INDEPENDENCE

Device independence is an important consideration when programming the System/360. The ability to request input/output operations without regard for the physical characteristics of the I/O devices makes it possible for you to write one program that will fulfill a variety of needs. Device independence may be useful for:

- Accepting data from a number of recording devices, e.g., 2311 disk pack, 7- or 9-track magnetic tape, or unit record equipment. This situation could arise when several types of data acquisition devices are feeding a centralized complex.

- Observing constraints imposed by the availability of input/output devices, i.e., devices on order have not been installed.

- Assembling, testing, and debugging on one System/360 configuration and processing on a different configuration, e.g., a 2311 direct access device can be used as a substitute for several magnetic tape units.

Device independence is clearly a valuable concept -- one that is not difficult to achieve, but which requires some planning and

forethought. There are two areas of planning necessary to achieve device independence -- system generation considerations and programming considerations.

SYSTEM GENERATION CONSIDERATIONS

The user of the operating system can provide for device independence when the system is generated. This is achieved by generating a system that meets not only the current input/output configuration requirements but includes anticipated device attachments. Creating such a system entails looking ahead at expected delivery of input/output devices and, during system generation, constructing in advance the necessary control blocks and tables. Thus, when the devices are delivered, they need only be physically attached. The operating system recognizes the devices without modification. During the interim, unconnected devices must be placed off-line. This is accomplished by a VARY command issued by the operator.

When new device attachments cannot be fully anticipated, new devices can be added by performing an I/O device generation. This is a limited type of system generation that enables the user to change his I/O configuration without regenerating other parts of the system.

Effecting a smooth transition to new input/output devices must not be construed to mean the inclusion of unsupported devices. This discussion is limited to add-on or substitution device independence. When support for new devices is provided, a new system will have to be generated. A complete description of system generation techniques is contained in the publication IBM System/360 Operating System: System Generation.

PROGRAMMING CONSIDERATIONS

Each of the data set organizations -- partitioned, indexed sequential, and direct -- requires the use of a direct access device. Device independence is meaningful, then, only in terms of a sequentially organized data set, that is, in a data set where one block of data follows another, thus allowing input or output to be on magnetic tape, direct access, card read/punch, or printer.

Your program will be device-independent if you do two things:

- Omit all device-dependent macro instructions or macro instruction parameters from your program.
- Defer specifying any required device-dependent parameters until the program is ready for execution. That is, supply the parameters on your data definition (DD) statement.

In examining the following list of macro instructions, consider only the logical layout of your data record without regard for the type of device used. Also, consider that any reference to a direct access volume is to be treated like magnetic tape, i.e., you must create a new data set rather than attempt to update.

OPEN
specify INPUT, OUTPUT, INOUT, or OUTIN. The parameters RDBACK and UPDATE are device dependent and cause an abnormal termination if directed to a different device type.

READ
specify forward reading only (SF).

WRITE
specify forward writing only (SF); use only to create new records.

PUTX
use only output mode.

NOTE/POINT
valid for both magnetic tape and direct access volumes.

BSP
valid for magnetic tape or direct access volumes. However, its use would be an attempt to perform device-dependent action.

CNTRL/PRTOV
device dependent

DCB Subparameters

MACRF
specify R/W or G/P. Processing mode can also be indicated.

DEVD
specify DA if any direct access device is apt to be used. Magnetic tape and unit record equipment data control blocks will fit in the area provided during assembly. Specify unit record devices only if you expect never to change to tape or direct access devices. Key length (KEYLEN) can be specified on the DD statement if necessary.

RECFM, LRECL, BLKSIZE
these can be specified in the DD statement. However, you must consider maximum record size for specific devices. Also, track overflow cannot be specified unless supported.

DSORG
specify sequential (PS/PSU).

OPTCD
device dependent; specify in the DD statement.

SYNAD
any device-dependent error checking is automatic. Generalize your routine so that no device-dependent information is required.

CHAINED SCHEDULING FOR I/O OPERATIONS

To accelerate the input/output operations required for a data set, the operating system provides a technique called chained scheduling. When requested, the system bypasses the normal I/O routines and dynamically chains several input/output operations together. A series of separate read or write operations, functioning with chained scheduling, is issued to the computing system as one continuous operation. The program-controlled interruption (PCI) flag in the CCWs is used for synchronization of the I/O operations.

The I/O performance is increased by reducing both the CPU time and channel start/stop time required to transfer data between main and secondary storage. The effects of rotational delay are also reduced since several successive blocks, requested separately, can be retrieved in a single rotation. Chained scheduling can be used only with simple buffering. Each data set for which chained scheduling is specified must be assigned at least two, and preferably three, buffers.

A request for chained scheduling will be ignored and normal scheduling used if any of the following are encountered when the data control block is opened:

- BDAM CREATE, i.e., MACRF=(WL).
- Track overflow.
- Operand of the OPEN macro instruction specifies UPDAT.
- Exchange buffering.
- CNTRL macro instruction to be used.
- Device type is paper tape reader.

When chained scheduling is being used, the automatic skip feature of the PRTOV macro instruction for the printer will not function. Format control must be achieved by ASCII or machine control characters. When using undefined length records with QSAM, the DCBLRECL field represents the block size instead of the actual record length.

Chained scheduling is most valuable for programs that require extensive input and output operations. Because a data set using chained scheduling may monopolize available time on a channel, separate channels should be assigned, if possible, when more than one data set is to be processed.

CREATING A SEQUENTIAL DATA SET

As discussed earlier, a processing program should be developed using factors that are constant. To provide for as much flexibility as possible, variable factors should be specified at execution time. For that reason, the following problem examples are generalized as much as possible. They are neither exhaustive nor intended as complete examples. Rather they are presented as introductory sequences.

Since the basic access technique for sequential processing is usually used to create a partitioned data set or a direct data set, examples of the READ/WRITE macro instructions are deferred for discussion in those areas. There is no other reason, however, for them not to be used in place of the queued access macro instructions where automatic blocking and anticipatory buffering are not required.

Tape-to-Print, Move Mode -- Simple Buffering: In this problem the GET-move and PUT-move require two movements of the data records. If the record length (LRECL) does not change in processing, only one move is necessary; you can process the record in the input buffer segment. A GET locate can be used to provide a pointer to the current segment.

```
          ...
          OPEN      (INDATA,,OUTDATA,(OUTPUT))
NEXTREC   GET       INDATA,WORKAREA          Move Mode
          AP        NUMBER,=P'1'
          UNPK      COUNT,NUMBER             Record count adds 6
          PUT       OUTDATA,COUNT            bytes to each record
          B         NEXTREC
TAPERROR  SYNADAF   ACSMETH=QSAM             Control program returns mess-
          LA        0,68(0,1)                age address in register 1.
          ST        14,SAVE14                SYNAD routine prints part of
          PUT       OUTDATA,(0)              the message (beginning with
          SYNADRLS                           the unit number) as a 56-byte
          L         14,SAVE14                fixed-length record. It then
          RETURN                             returns to the control
ENDJOB    CLOSE     (INDATA,,OUTDATA)        program.
          ...
COUNT     DS        CL6
WORKAREA  DS        CL50
NUMBER    DC        PL4'0'
SAVE14    DS        F
INDATA    DCB       DDNAME=INPUTDD,DSORG=PS,MACRF=(GM),EROPT=ACC,        C
               SYNAD=TAPERROR,EODAD=ENDJOB
OUTDATA   DCB       DDNAME=OUTPUTDD,DSORG=PS,MACRF=(PM),EROPT=ACC
```

Tape-to-Print, Locate Mode -- Simple Buffering: This problem is similar to the previous one. However, since there is no change in the record length, the records can be processed in the input buffer. Only one move of each data record is required.

```
          ...
          OPEN      (INDATA,,OUTDATA,(OUTPUT),ERRORDCB,(OUTPUT))
NEXTREC   GET       INDATA                   Locate Mode
          LR        2,1                      Save Pointer
          AP        NUMBER,=P'1'
          UNPK      0(6,2),NUMBER            Process in Input Area
          PUT       OUTDATA                  Locate Mode
          MVC       0(50,1),0(2)             Move record to output buffer
          B         NEXTREC
TAPERROR  SYNADAF   ACSMETH=QSAM             Message address in register 1
          ST        2,SAVE2                  Save register 2 contents
          L         2,8(0,1)                 Load pointer to input buffer
          MVC       8(70,1),50(1)            Shift nonblank message fields
          MVC       78(50,1),0(2)            Add input record to message
          LR        0,1                      Load address of message
          LR        2,14                     Save return address
          PUT       ERRORDCB,(0)             Print message (move mode)
          SYNADRLS                           Release message and save area
          LR        14,2                     Restore return address
          L         2,SAVE2                  Restore register 2
          RETURN                             Return to control program
ENDJOB    CLOSE     (INDATA,,OUTDATA,,ERRORDCB)
          ...
NUMBER    DC        PL4'0'
INDATA    DCB       DDNAME=INPUTDD,DSORG=PS,MACRF=(GL),EROPT=ACC,        C
               SYNAD=TAPERROR,EODAD=ENDJOB
OUTDATA   DCB       DDNAME=OUTPUTDD,DSORG=PS,MACRF=(PL),EROPT=ACC
ERRORDCB  DCB       DDNAME=SYSOUTDD,DSORG=PS,MACRF=(PM),RECFM=V,         C
               BLKSIZE=128,LRECL=124,EROPT=ACC
SAVE2     DS        F
```

Tape-to-Print, Substitute Mode -- Exchange Buffering: Although the initial problem is the same, the solution described here takes advantage of two facilities: exchange buffering, which eliminates the

need to move the data record; and direct reference to individual fields within a record through the use of a dummy control section (DSECT). The use of the DSECT allows symbolic reference to be made for storage-to-storage operations; therefore, the length attributes need not be explicitly stated.

```
          ...
          OPEN      (INDATA,,OUTDATA,(OUTPUT),ERRORDCB,(OUTPUT))
          LA        0,GIVEAWAY                Set up for first buffer
NEXTREC   GET       INDATA,(0)                Substitute Mode
          LR        2,1                       Pointer to next record
          USING     RECORD,2                  Establish address of DSECT
          AP        NUMBER,=P'1'
          UNPK      COUNT,NUMBER
          PUT       OUTDATA,RECORD            Substitute Mode
          LR        0,1                       Exchange work area
          B         NEXTREC
TAPERROR  SYNADAF   ACSMETH=QSAM              SYNAD routine is same
          ...                                 as in previous example
ENDJOE    CLOSE     (INDATA,,OUTDATA,,ERRORDCB)
          ...
          DS        0D
GIVEAWAY  DS        CL50
NUMBER    DC        PL4'0'
INDATA    DCB       DDNAME=INPUTDD,DSORG=PS,MACRF=(GT),BFTEK=E,BFALN=D, C
                EROPT=ACC,SYNAD=TAPERROR,EODAD=ENDJOB
OUTDATA   DCB       DDNAME=OUTPUTDD,DSORG=PS,MACRF=(PT),BFTEK=E,BFALN=D,C
                EROPT=ACC
          ...
RECORD    DSECT
COUNT     DS    ZL6
RESTOFIT  DS    CL44
```

Processing a Partitioned Data Set

A partitioned data set is divided into sequentially organized members made up of one or more records (see Figure 22). Each member has a unique name, one to eight characters long, stored in a directory. The records of a given member are stored or retrieved sequentially.

The main advantage of using a partitioned data set is that you can retrieve any individual member once the data set is opened. For example, a program library can be stored as a partitioned data set, each member of which is a separate program or subroutine. The individual members can be added or deleted as required. When a member is deleted, only the member name is removed from the directory; the space used by the member cannot be reused until the data set is reorganized.

The directory, a series of records at the beginning of the data set, contains an entry for each member. Each directory entry contains the member name and the starting location of the member within the data set, as shown in Figure 29. The directory entries are arranged in alphameric collating sequence by name. In addition, you can specify up to 62 characters of information in the entry.

The track address of each member is recorded by the system as a relative track within the data set rather than as an absolute track address. Thus, an entire data set can be moved without changing the relative track addresses. The data set can be considered as one continuous set of data tracks regardless of how the space was actually

allocated. If there is not sufficient space available in the directory for an additional entry, or not enough space available within the data set for an additional member, no new members can be stored.

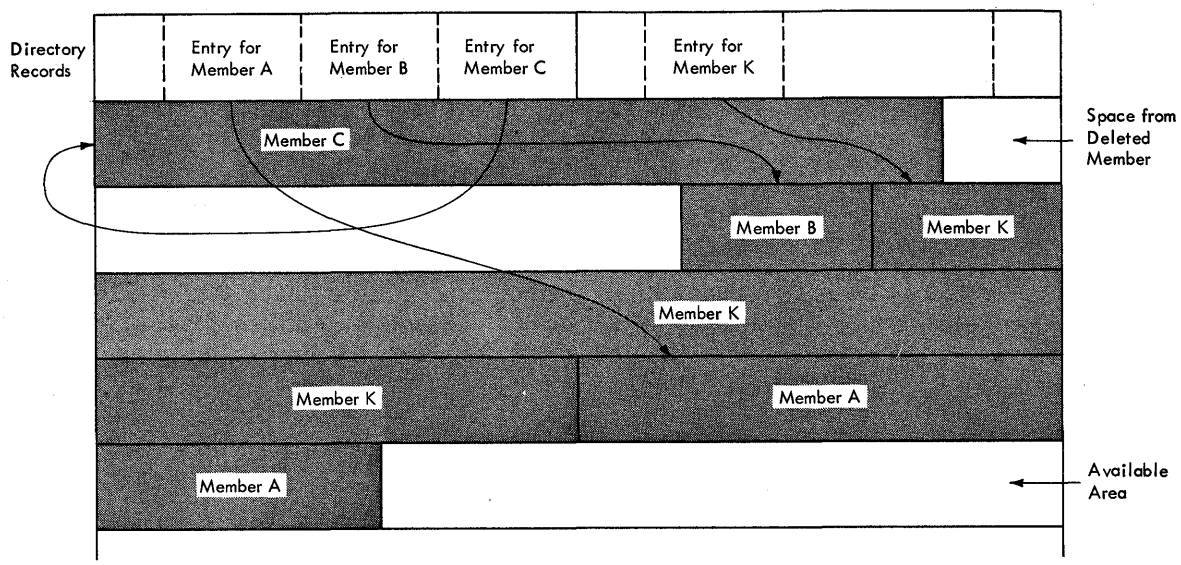


Figure 29. A Partitioned Data Set

PARTITIONED DATA SET DIRECTORY

The directory of a partitioned data set occupies the beginning of the area allocated to the data set on a direct access volume. It is searched and maintained by the FIND and STOW macro instructions. The directory consists of member entries arranged in ascending order according to the binary value of the member name or alias.

Member entries vary in length and are blocked into 256-byte blocks. Each block contains as many complete entries as will fit in a maximum of 254 bytes; any remaining bytes are left unused and are ignored. Each directory block contains a 2-byte count field that specifies the number of active bytes in a block. As shown in Figure 30, each block is preceded by a hardware-defined key field containing the name of the last member entry in the block, i.e., the member name with the highest binary value.

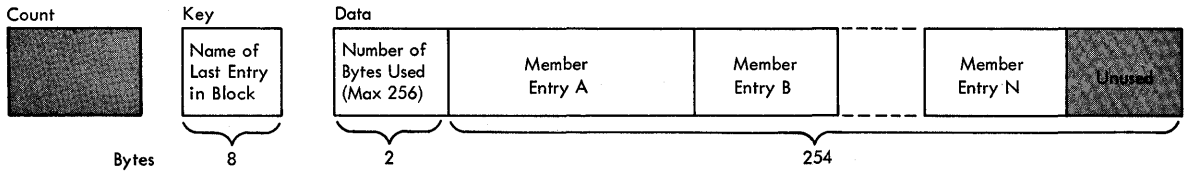


Figure 30. A Partitioned Data Set Directory Block

Each member entry contains a member name or alias; there can be up to 16 aliases (alternate names) for each member. Each entry also contains the relative track address of the member and a count field, as shown in Figure 31. In addition, it may contain a user data field. The last entry in the last directory block has a name field of maximum binary value -- all ones.

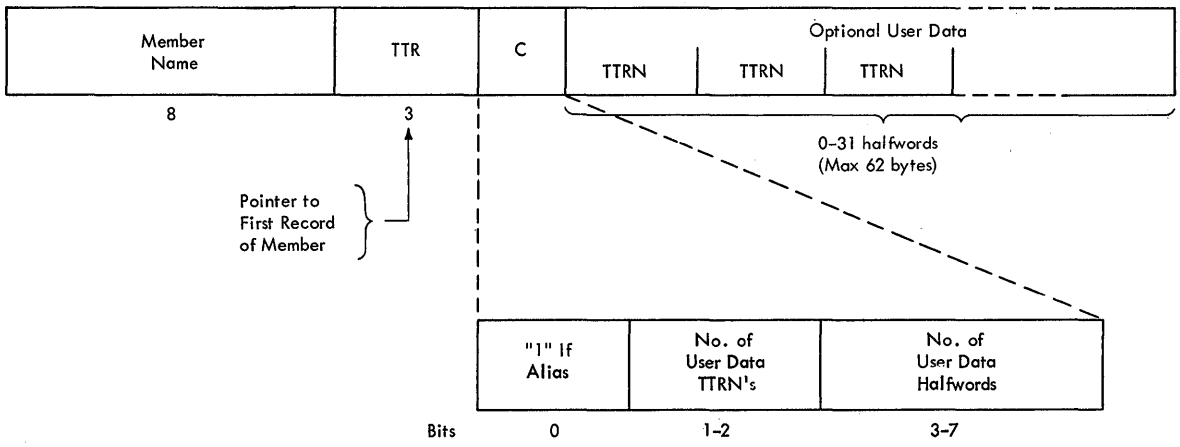


Figure 31. A Partitioned Data Set Directory Entry

NAME

specifies the member name or alias. It contains up to eight alphameric characters, left-justified and padded with blanks if necessary.

TTR

is a pointer to the first block of the member; TT is the relative track from the beginning of the data set, and R is the relative block number on that track.

Note: This pointer is created by adding one to the TTR for the last block of the previous member (which is an end-of-file mark). If track TT is full, the next block will begin at record one of track TT+1, and the pointer will be updated accordingly. The control program finds the block by searching in multitrack mode using TT(R-1) as a search argument.

C

specifies the number of halfwords contained in the user data field. It may also contain additional information about the user data field, as shown below:

```
Bits   0 1-2  3-7
      ┌─┬──┬─────┐
      └─┴──┴─────┘
```

0 when set to 1, indicates that the NAME field contains an alias.

1-2 specifies the number of pointers to locations within the member.

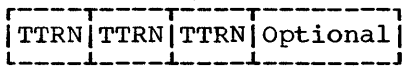
A maximum of three pointers is allowed in the user data field. Additional pointers may be contained in a record referred to as a note list discussed below. The pointers can be updated automatically if the data set is moved or copied by a utility program such as the IEHMOVE utility program. The data set must be marked "unmovable" under the following conditions:

- More than three pointers are used in the user data field.
- The pointers in the user data field or note list do not conform to the standard format.
- The pointers are not placed first in the user data field.
- Any direct access addresses (absolute or relative) are embedded in any data blocks or in another data set that refers to this data set.

3-7 contains a binary value indicating the number of halfwords of user data. This number must include the space used by pointers in the user data field.

The user data field contains variable user data provided as input to the STOW macro instruction. If pointers to locations within the member are provided, they must be four bytes long and placed first in the user data field. The user data field format is as follows:

User Data



TT is the relative track address of the note list or area to which you are pointing.

R is the relative block number on that track.

N is a binary value that indicates the number of additional pointers contained in a note list pointed to by the TTR. If the pointer is not to a note list, N=0.

A note list consists of additional pointers to blocks within the same member of a partitioned data set. If the existence of a note list was indicated as shown above, the list can be updated automatically when the data set is moved or copied by a utility program such as the IEHMOVE utility program. Each 4-byte entry in the note list has the following format:

TT is the relative track address of the area to which you are pointing.

R is the relative block number on that track.

X is available for any use.

To place the note list in the partitioned data set, you must use the WRITE macro instruction. After checking the write operation, use the NOTE macro instruction to determine the address of the list and place that address in the user data field of the directory entry.

PROCESSING A MEMBER OF A PARTITIONED DATA SET

Because a member of a partitioned data set is sequentially organized, it is processed in the same manner as a sequential data set. Either the basic or queued access technique can be used. However, you cannot alter the directory when using the queued technique.

In order to locate a member or to process the directory, several macro instructions are provided by the operating system. The BLDL macro instruction can be used to structure a list of directory entries in main storage; the FIND macro instruction locates a member of the data set for subsequent processing; the STOW macro instruction adds or deletes a member name in the directory. To use these macro instructions, you must specify DSORG=PO or POU in the DCB macro instruction. Before issuing a FIND, BLDL, or STOW macro instruction, you must check all preceding input/output operations for completion.

BLDL -- Construct a Directory Entry List

The BLDL macro instruction is used to place directory information in main storage. The data is placed in a "build" list constructed by you before the BLDL macro instruction is issued. The format of the list is similar to the directory. For each member name in the list, the system supplies the address of the member and any additional information contained in the directory entry.

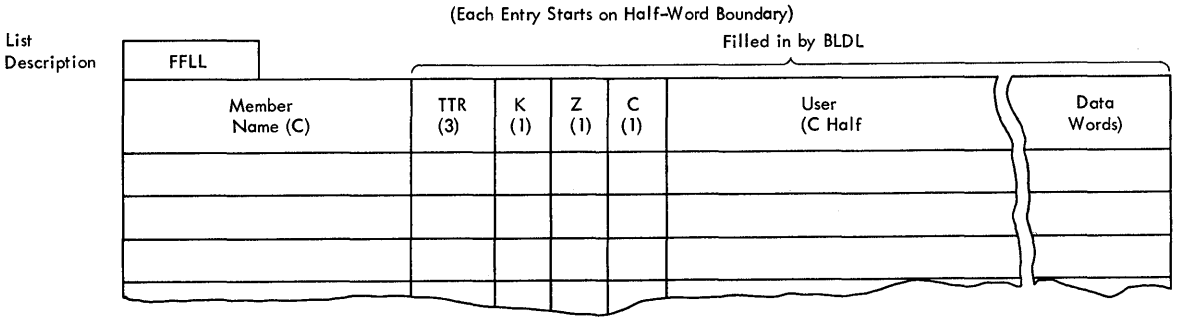
You can optimize retrieval time by directing a subsequent FIND macro instruction to the build list rather than the directory to locate the member to be processed.

The build list, as shown in Figure 32, must be preceded by a 4-byte list description that indicates the number of entries in the list and the length of each entry (14 to 76 bytes). The first eight bytes of each entry contain the member name or alias. The next six bytes must be available to contain the starting address of the member plus some control data. If additional information is to be supplied from the directory, up to 62 bytes can be reserved.

FIND -- Position to a Member

To determine the starting address of a specific member you must issue a FIND macro instruction. If you want to find only one member, the function is performed automatically when you specify the data set name and member name in the related DD statement. The system places the correct address in the data control block so that a subsequent GET/READ macro instruction will begin processing at that point.

There are two ways in which the system can be directed to the desired member: you can specify the address of either an area containing the name of the member or an entry in a build list you have created. In the first case, the system searches the directory of the data set. If a build list is used, no search is required; the relative track address is determined from the list entry.



Programmer Supplies:
- FF = Number of member entries in list
- LL = Even no. giving byte length of each entry (minimum of 12)
- Member name = eight bytes, left-adjusted

BLDL Supplies:
- TTR = Member starting location
- K = If only data set = 0
  If concatenation = no.
- Z = Normally padding for boundary alignment
- C = Same C field from directory. Gives no. of user data halfwords
- User data: as much as will fit in entry

Figure 32. Build List Format

STOW -- Alter a Directory Entry

Unless you are adding members to a partitioned data set one at a time, you must issue a STOW macro instruction to enter the member name in the directory. When adding a single member, the STOW function is performed automatically when the data set is closed.

You can also use the STOW macro instruction to delete, replace, or change a name in the directory, as well as store additional information with the directory entry. Since an alias can also be stored in the directory in the same way, you should be consistent in altering all names associated with a given member. For example, if you replace a member, you must delete related aliases or change them so that they point to the new member. If you use STOW to change user data in the directory entry, you must also move the TTR of the member into the DCBRELAD.

If you do not use the STOW macro instruction before closing a partitioned data set that you have written, your CLOSE request causes the system to issue a STOW macro instruction. If you specify DISP=MOD, the system issues a STOW macro instruction with the replace option, causing replacement of an entry in the directory. If you specify DISP=NEW or DISP=OLD and the member does not exist, the system issues a STOW macro instruction with the add option, causing addition of an entry to the directory. If you specify DISP=OLD and the member already exists, the system issues a message to that effect.

CREATING A PARTITIONED DATA SET

If you have no need to add entries to the directory, i.e., the STOW and BLDL macro instructions will not be used, you can create a new data set and write the first member as follows:

- Code DSORG=PS or PSU in the DCB macro instruction.
- Indicate in the DD statement that the data is to be stored as a member of a new partitioned data set, i.e., DSNAME=name(membername) and DISP=NEW.
- Request space for the member and the directory in the DD statement.
- Process the member with an OPEN macro instruction, a series of PUT/WRITE macro instructions, and then a CLOSE macro instruction. A STOW macro instruction is issued automatically when the data set is closed.

As a result of these steps, the data set and its directory are created, the records of the member are written, and an entry is made in the directory.

To add additional members to the data set, follow the same procedure. However, a separate DD statement (with the space request omitted) is required for each member. The disposition should be specified as modify, DISP=MOD. The data set must be closed and reopened each time a new member is specified.

```
//PDSDD  DD        ---,DSNAME=MASTFILE(MEMBERK),SPACE=(TRK,(100,5,7)), C
//             DISP=(NEW,KEEP)
```

```
OUTDCB   DCB       ---,DSORG=PS,DDNAME=PDSDD,---
         ...
         OPEN      (OUTDCB,(OUTPUT))
         PUT   (or WRITE)
         ...
         CLOSE     (OUTDCB)                    Automatic Stow
```

To take full advantage of the STOW macro instruction, and thus the BLDL and FIND macro instructions in future processing, you can provide additional information with each directory entry. This is accomplished by using the basic access technique, which also allows you to process more than one member without closing and reopening the data set, as follows:

- Request space in the DD statement for the members and the directory.
- Define DSORG=PO or POU in the DCB macro instruction.
- WRITE (and CHECK) the member records.
- NOTE the location of any note list written within the member, if there is a note list.
- When all the member records have been written, issue a STOW macro instruction to enter the member name, its location pointer, and any additional data in the directory.
- Continue to WRITE, CHECK, NOTE, and STOW until all the members of the data set and the directory entries have been written.

```
//PDSDD DD  --,DSNAME=MASTFILE,SPACE=(TRK,(100,5,7)),DISP=MOD
```

```
OUTDCB    DCB  --,DSORG=PO,DDNAME=PDSDD,--
          OPEN (OUTDCB,(OUTPUT))
          WRITE   *
          CHECK   First record of member.
          NOTE
          WRITE
          CHECK   Remaining records of member.
          (NOTE)  Only NOTE first record of a subgroup within member.
          WRITE
          CHECK   Write note lists at end of each
          NOTE    subgroup.
          STOW    Member entry in directory after all records and note
                  lists are written.
Repeat from * for each additional member
          CLOSE (OUTDCB)
```

RETRIEVING A MEMBER

To retrieve a specific member from a partitioned data set, either the basic or queued access technique can be used as follows:

- Code DSORG=PS or PSU in the DCB macro instruction.

- Indicate in the DD statement that the data is a member of an existing partitioned data set, i.e., DSNAME=name(membername) and DISP=OLD.

- Process the member with an OPEN macro instruction, a series of GET/READ macro instructions, and then a CLOSE macro instruction.

When you code RECFM for the DCB macro instruction, note that standard record format (S) is not allowed for BPAM. If you include S in your record format specification, permanent input/output errors (no-record-found condition) can occur.

When your program is executed, the directory is searched automatically and the location of the member is placed in the data control block.

```
-------------------------------------------------------------------------
//PDSDD  DD     --,DSNAME=MASTFILE(MEMBERK),DISP=OLD
-------------------------------------------------------------------------
INDCB    DCB    --,DSORG=PS,DDNAME=PDSDD,--
         OPEN   (INDCB)      Automatic Find
         GET (or READ)
         CLOSE  (INDCB)
```

In order to process several members without closing and reopening, or to take advantage of additional data in the directory, the following technique should be used:

- Code DSORG=PO or POU in the DCB macro instruction.

- Build a list (BLDL) of needed member entries from the directory.

- Indicate in the DD statement the data set name of the partitioned data set, i.e., DSNAME=name, and DISP=OLD.

- Use the FIND or POINT macro instruction to prepare for reading the member records.

- The records may be read from the beginning of the member, or a note list may be read first, to obtain additional locations that POINT to sub-categories within the member.

- READ (and CHECK) the records until all those required have been processed.

- POINT to additional categories, if required, and READ the records.

- Repeat this procedure for each member to be retrieved.

```
//PDSDD  DD     --,DSNAME=MASTFILE,DISP=OLD
```

```
INDCB    DCB    --,DSORG=PO,DDNAME=PDSDD,--
         OPEN   (INDCB)
         BLDL         Build a list of selected member names in main
                      storage.
         FIND (or POINT)
         READ         *Read note list.
         CHECK
         POINT        Locate subgroup by using note list.
         READ
         CHECK        Read member records
Repeat from * for each additional member.
         CLOSE  (INDCB)
```

UPDATING A MEMBER

A member of a partitioned data set can be updated in place, or can be deleted and rewritten as a new member.

UPDATING IN PLACE

When you update in place, you read records, process them, and write them back to their original positions without destroying the remaining records on the track. The following rules apply:

- You must specify the update option (UPDAT) in the OPEN macro instruction. To perform the update, you can use only the READ, WRITE, CHECK, NOTE, POINT, FIND, and BLDL macro instructions.
- You cannot use chained scheduling.
- You cannot delete any record or change its length; you cannot add new records.

A record must be retrieved by a READ macro instruction before it can be updated by a WRITE macro instruction. Both macro instructions must be "execute" forms that refer to the same data event control block (DECB); the DECB must be provided by a "list" form. (The execute and list forms of the READ and WRITE macro instructions are described in the publication IBM System/360 Operating System: Supervisor and Data Management Macro Instructions.)

Updating With Overlapped Operations: To overlap input/output and CPU activity, you can start several read or write operations before checking the first for completion. You cannot overlap read and write operations, however, as operations of one type must be checked for completion before operations of the other type are started or resumed. Note that each concurrent read or write operation requires a separate channel program, and also a separate DECB. If a single DECB were used for successive read operations, only the last record read could be updated.

In the following example, overlap is achieved by having a read or write request outstanding while each record is being processed. Note the use of execute- and list-form macro instructions, identified by the operands MF=E and MF=L.

```
//PDSDD    DD     DSNAME=MASTFILE(MEMBERK),DISP=OLD,---
UPDATDCB  DCB    DSORG=PS,DDNAME=PDSDD,MACRF=(R,W),NCP=2,EODAD=FINISH
          READ   DECBA,SF,UPDATDCB,AREAA,MF=L   Define DECBA
          READ   DECBB,SF,UPDATDCB,AREAB,MF=L   Define DECBB
AREAA     DS     ---                            Define buffers
AREAB     DS     ---
          ...
          OPEN   (UPDATDCB,UPDAT)       Open for update
          LA     2,DECBA                Load DECB addresses
          LA     3,DECBB
READRECD  READ   (2),SF,MF=E            Read a record
NEXTRECD  READ   (3),SF,MF=E            Read the next record
          CHECK  (2)                    Check previous read operation
          (If update is required, branch to R2UPDATE)
          LR     4,3                    If no update is required,
          LR     3,2                      switch DECB addresses in
          LR     2,4                      registers 2 and 3
          B      NEXTRECD                 and loop

* In the following statements, "R2" and "R3" refer to the records
* that were read using the DECBs whose addresses are in registers
* 2 and 3, respectively.  Either register may point to either
* DECBA or DECBB.

R2UPDATE  CALL   UPDATE,((2))           Call routine to update R2
          CHECK  (3)                    Check read for next record (R3)
          WRITE  (2),SF,MF=E            Write updated R2
          (If R3 requires an update, branch to R3UPDATE)
          CHECK  (2)                    If R3 requires no update, check
          B      READRECD                 write for R2 and loop
R3UPDATE  CALL   UPDATE,((3))           Call routine to update R3
          WRITE  (3),SF,MF=E            Write updated R3
          CHECK  (2)                    Check write for R2
          CHECK  (3)                    Check write for R3
          B      READRECD               Loop
FINISH    CLOSE  (UPDATDCB)             End-of-data exit routine
          ...
```

REWRITING A MEMBER

There is no actual update option that can be used to add or extend records in a partitioned data set. If you want to extend or add a record within a member, you must rewrite the complete member in another area of the data set. Since space is allocated when the data set is created, there is no need to request additional space. Note, however, that a partitioned data set must be contained on one volume. If sufficient space has not been allocated, the data set must be reorganized by the IEBUPDTE utility program.

When you rewrite the member, you must provide two data control blocks; one for input and one for output. Both DCB macro instructions can refer to the same data set, i.e., only one DD statement is required.

You can reflect the change in location of the member either automatically, by indicating a disposition of OLD, or by using the STOW macro instruction. Although the old member is, in effect, deleted, its space cannot be reused until the data set is reorganized.

Processing an Indexed Sequential Data Set

An indexed sequential data set allows you a great deal of flexibility in the operations you can perform. The data set can be read or written sequentially; individual records can be processed in any order; records can be deleted; or new records can be added. The system automatically locates the proper position in the data set for new records and makes any necessary adjustments when records are deleted. This flexibility is possible due to the inherent organization of the data set.

Although the queued and basic access techniques can be used to process an indexed sequential data set, each has separate and distinct functions. The queued access technique must be used to create the data set. It can also be used to sequentially process or update the data set and to add records to the end of the data set. The basic access technique can be used to insert new records between records already in the data set. It can also be used to update the data set directly.

INDEXED SEQUENTIAL DATA SET ORGANIZATION

The records in an indexed sequential data set are arranged according to the collating sequence of a key field in each record. Each block of records is preceded by a key field that corresponds to the key of the last record in the block.

An indexed sequential data set resides on direct access storage devices and can occupy up to three different areas:

- Prime Area -- This area contains data records and related track indexes. It exists for all ISAM data sets.
- Overflow Area -- This area contains overflow from the prime area when new data records are added. It is optional.
- Index Area -- This area contains master and cylinder indexes associated with the data set. It exists for a data set that has a prime area occupying more than one cylinder.

The indexes of an ISAM data set are analogous to the index card file in a library. For example, if the library user knows the name of the book or the author, he can look in the index card file and obtain a catalog number that will enable him to locate the book in the book files. He would then go to the shelves and proceed through each row until he found the shelf containing the book. Usually each row contains a sign to indicate the beginning and ending numbers of all books in that particular row. Thus, as he proceeded through the rows, he would compare the catalog number obtained from the index with the numbers posted on each row. Upon locating the proper row, he would then search that row for the shelf that contained the book. Then he would look at the individual book numbers on that shelf until he found the particular book.

ISAM uses the indexes in much the same way to locate records in an indexed sequential data set. The operating system provides both the queued and basic access techniques to process an indexed sequential data set. The queued access technique is used to create the data set and add records to the end. It can also be used to sequentially process or update the records. The basic access technique is used to read or update records and to insert new records at any place in the data set.

As the records are written in what is referred to as the prime area of the data set, the system accounts for the records contained on each track in a track index area. Each entry in the track index identifies the key of the last record on each track. There is a track index for each cylinder in the data set. If more than one cylinder is used, the system develops a higher level index called a cylinder index. Each entry in the cylinder index identifies the key of the last record in the cylinder. To increase the speed of searching the cylinder index, you can request that a master index be developed for a specified number of cylinders, as shown in Figure 33.

Rather than reorganize the whole data set when records are added, you can request that space be allocated for additional records in what is called an overflow area.

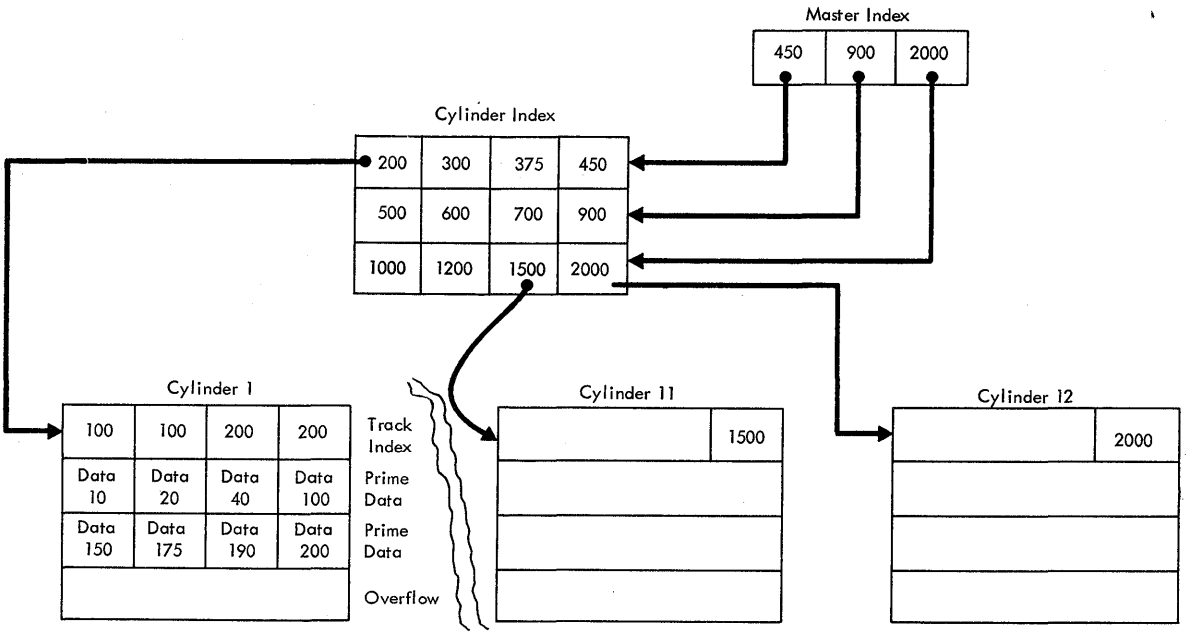


Figure 33. Indexed Sequential Data Set Organization

PRIME AREA

Records are written in the prime area when the data set is created or updated. The portion of Figure 33 labeled Cylinder 1 illustrates the initial structure of the prime area. Although the prime area can extend across several noncontiguous areas of the volume, all the records are written in key sequence. Each record must contain a key; the system automatically writes the key of the highest record preceding each block.

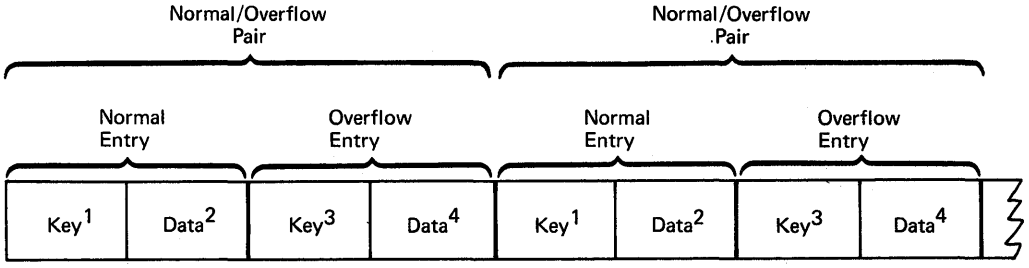
When the ABSTR option of the SPACE parameter of the DD statement is used to generate a multivolume prime area, the VTOC on the second volume and on all succeeding volumes must be contained within cylinder zero of the volume.

INDEX AREAS

The operating system generates track and cylinder indexes automatically. Up to three levels of master indexes are created if requested.

Track Index: This is the lowest level of index and is always present. There is one track index for each cylinder in the prime area; it is written on the first tracks of the cylinder that it indexes.

The index consists of a series of paired entries, that is, of a normal entry and an overflow entry for each prime track. The normal entry contains the key of the highest record on the track and the home address of the prime track. The overflow entry is originally the same as the normal entry. (This is why 100 appears twice on the track index for cylinder 1 in Figure 33.) The overflow entry is changed when records are added to the data set. Then the overflow entry contains the key of the highest overflow record and the address of the lowest overflow record logically associated with the particular prime track. Figure 34 shows the format of a track index.



[1]Normal key = key of the highest record on the prime data track.

[2]Normal data = address of the prime data track.

[3]Overflow key = key of the highest overflow record logically associated with the prime data track.

[4]Overflow data = address of the lowest overflow record logically associated with the prime data track.

Notes:

- If there are no overflow records, overflow key and data entries are the same as normal key and data entries.
- This figure is a logical representation only; that is, it makes no attempt to show the physical size of track index entries.

● Figure 34. Format of Track Index Entries

If all the tracks allocated for the prime data area are not used, their entries in the index are "flagged" as inactive. The last entry of each track index is a dummy entry indicating the end of the index. When fixed-length record format has been specified, the remainder of the last track used for a track index contains prime data records if there is room for them.

Each index entry has the same format. It is an unblocked, fixed-length record consisting of a count, a key, and a data area. The length of the key corresponds to the length of the key area in the record to which it points. The data area is always ten bytes long. It contains the full address of the track or record to which the index points, as well as the level of the index and the entry type.

Cylinder Index: For every track index created, the system generates a cylinder index entry. There is one cylinder index for a data set, each entry of which points to a track index. Since there is one track index per cylinder, there is one cylinder index entry for each cylinder in the prime data area, except for a one-cylinder prime area. As with track indexes, inactive entries are created for any unused cylinders in the prime data area.

Master Index: As an optional feature, the operating system creates, at your request, a master index. Each entry in the master index points to a cylinder index track. This facility avoids a serial search through a large cylinder index.

You can specify the number of entries that are to be included in each master index. For example, if you indicate that you want a master index created for every three tracks of cylinder index entries, a master index is created if the cylinder index exceeds three tracks. If your data set is extremely large, a higher level master index is created if the first level master index exceeds three tracks. This procedure continues up to three levels of master indexes.

OVERFLOW AREAS

As records are added to an indexed sequential data set, space is required to contain those records that will not fit on the prime data track on which they belong. You can request that a number of tracks be set aside as a cylinder overflow area to contain overflows from prime tracks in each cylinder. An advantage of using cylinder overflow areas is a reduction of search time required to locate overflow records. A disadvantage is that there will be unused space if the additions are unevenly distributed throughout the data set.

Instead of, or in addition to, cylinder overflow areas, you can request an independent overflow area. Overflow from anywhere in the prime data area is placed in a specified number of cylinders reserved solely for overflow records. An advantage of having an independent overflow area is a reduction in unused space reserved for overflow. A disadvantage is the increased search time required to locate overflow records in an independent area.

If you request both cylinder overflow and independent overflow, cylinder overflow is used first.

It is a good practice to request cylinder overflow areas large enough to contain a reasonable number of additional records and an independent overflow area to be used as the cylinder overflow areas are filled.

ADDING RECORDS TO AN INDEXED SEQUENTIAL DATA SET

Either the queued or the basic access technique may be used to add records to an indexed sequential data set. A record to be inserted between records already in the data set must be inserted by the basic access method using WRITE KN (key new). Records added to the end of a data set, that is, records with successively higher keys, may be added to the overflow chain by the basic access method using WRITE KN (key new); or they may be added to the prime data area by the queued access technique using the PUT macro instruction.

INSERTING NEW RECORDS INTO AN EXISTING INDEXED SEQUENTIAL DATA SET

As you add records to an indexed sequential data set, the system inserts each record in its proper sequence according to the record key. The remaining records on the track are then moved up one position. If the last record does not fit on the track, it is written in the first available location in the overflow area. A 10-byte link field is added to the "bumped" record to connect it logically to the correct track. The proper adjustments are made to the track index entries. This procedure is illustrated in Figure 35.

Subsequent additions are written either on the prime track where they belong or as part of the overflow chain from that track. If the addition belongs after the last prime record on a track but before a previous overflow record from that track, it is written in the first available location in the overflow area. Its link field contains the address of the next record in the chain.

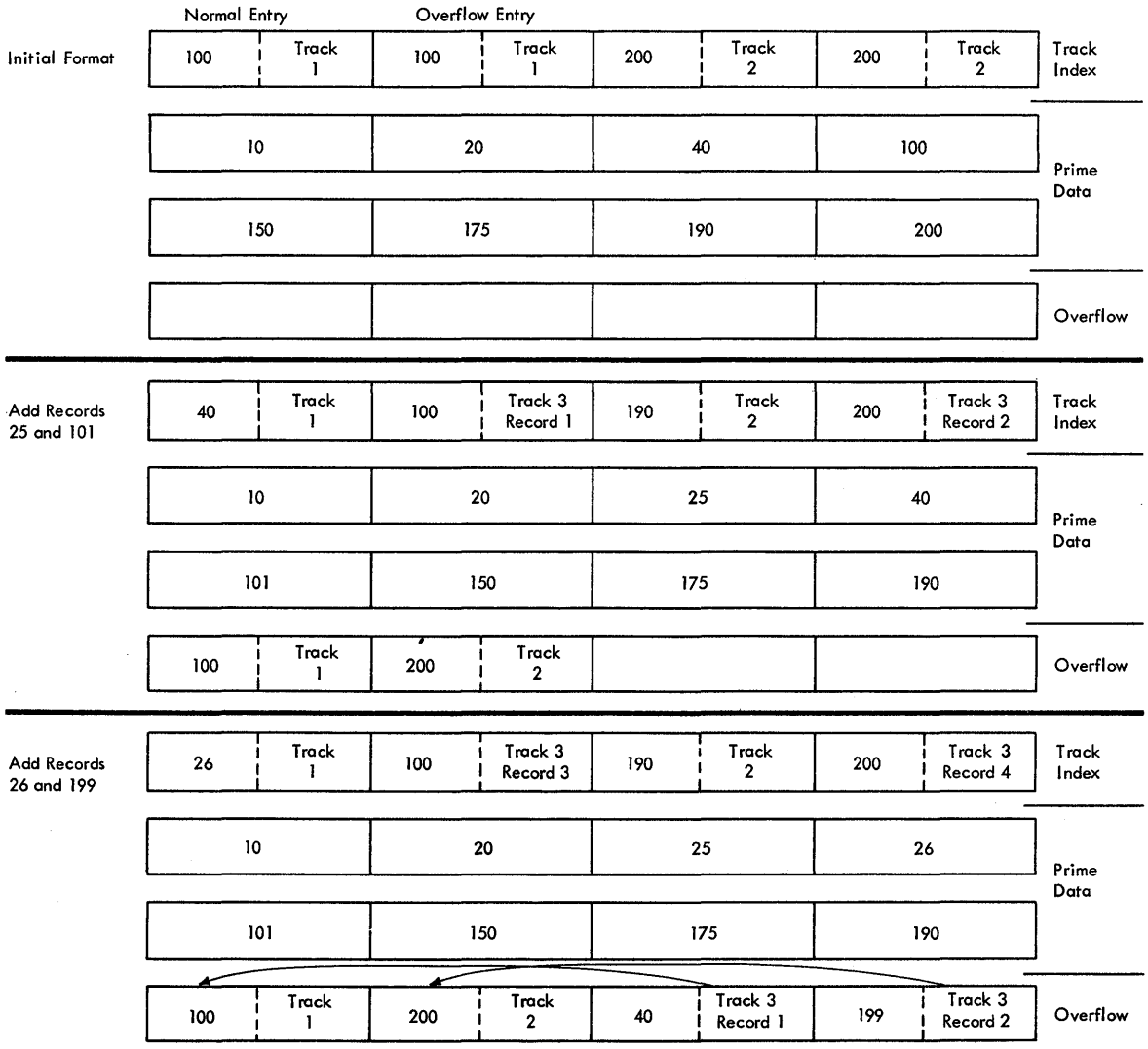


Figure 35. Adding Records to an Indexed Sequential Data Set

ADDING NEW RECORDS TO THE END OF AN INDEXED SEQUENTIAL DATA SET

Records added to the end of a data set, that is, records with successively higher keys, may be added by the basic access method using WRITE KN (key new), or by the queued access method using the PUT macro instruction (Resume Load). In either case records may be added to the prime data area. When you use the WRITE KN macro instruction, the record being added will be placed in the prime data area only if there is room for it on the prime data track containing the record with the highest key currently in the data set. If there is not sufficient room on that track, the record is placed in the overflow area and linked to that prime track via the overflow chain even though additional prime data tracks originally allocated have not been filled. When you use the PUT macro instruction (Resume Load), records

will be added to the prime data area until the space originally allocated is filled. Once this allocated prime area is filled, you can add records to the data set using WRITE KN, in which case they will be placed in the overflow area. You can add variable-length records to an indexed sequential data set only by using the WRITE KN macro instruction.

In order to add records with successively higher keys using the PUT macro instruction (Resume Load):

- The key of any record to be added must be higher than the highest key currently in the data set.
- The DD statement must specify DISP=MOD.
- The data set must have been successfully closed when it was created or when records were previously added using the PUT macro instruction.

You may continue to add fixed-length records in this manner until the original space allocated for prime data is exhausted.

When adding records to an indexed sequential data set using the PUT macro instruction (Resume Load), new entries are also made in the indexes. During Resume Load on a data set with a partially filled track and/or a partially filled cylinder, the track index entry and/or the cylinder index entry is overlaid when the track or cylinder is filled. If Resume Load abnormally terminates after these index entries have been overlaid, a subsequent Resume Load will get a sequence check when adding a key that is higher than the highest key at the last successful CLOSE but lower than the key in the overlaid index entry. When the SYNAD exit is taken for a sequence check, register 0 contains the address of the high key of the data set.

MAINTAINING AN INDEXED SEQUENTIAL DATA SET

An indexed sequential data set must be reorganized periodically for two reasons:

- The overflow area will eventually be filled.
- Additions increase the time required to locate records directly.

The frequency of reorganization depends on the activity of the data set and on your timing and storage requirements. There are two ways you can accomplish reorganizations:

- The data set can be written sequentially into another area of direct access storage or magnetic tape and then re-created in the original area.
- It can be reorganized in one pass by writing it directly into another area of direct access storage. In this case, the area occupied by the original data set cannot be used by the reorganized data set.

The operating system maintains statistics that are pertinent to reorganization. The statistics are written on the direct access volume and are available to you for checking. The information includes the number of cylinder overflow areas, the number of unused tracks in the independent overflow area, and the number of references to overflow records other than the first.

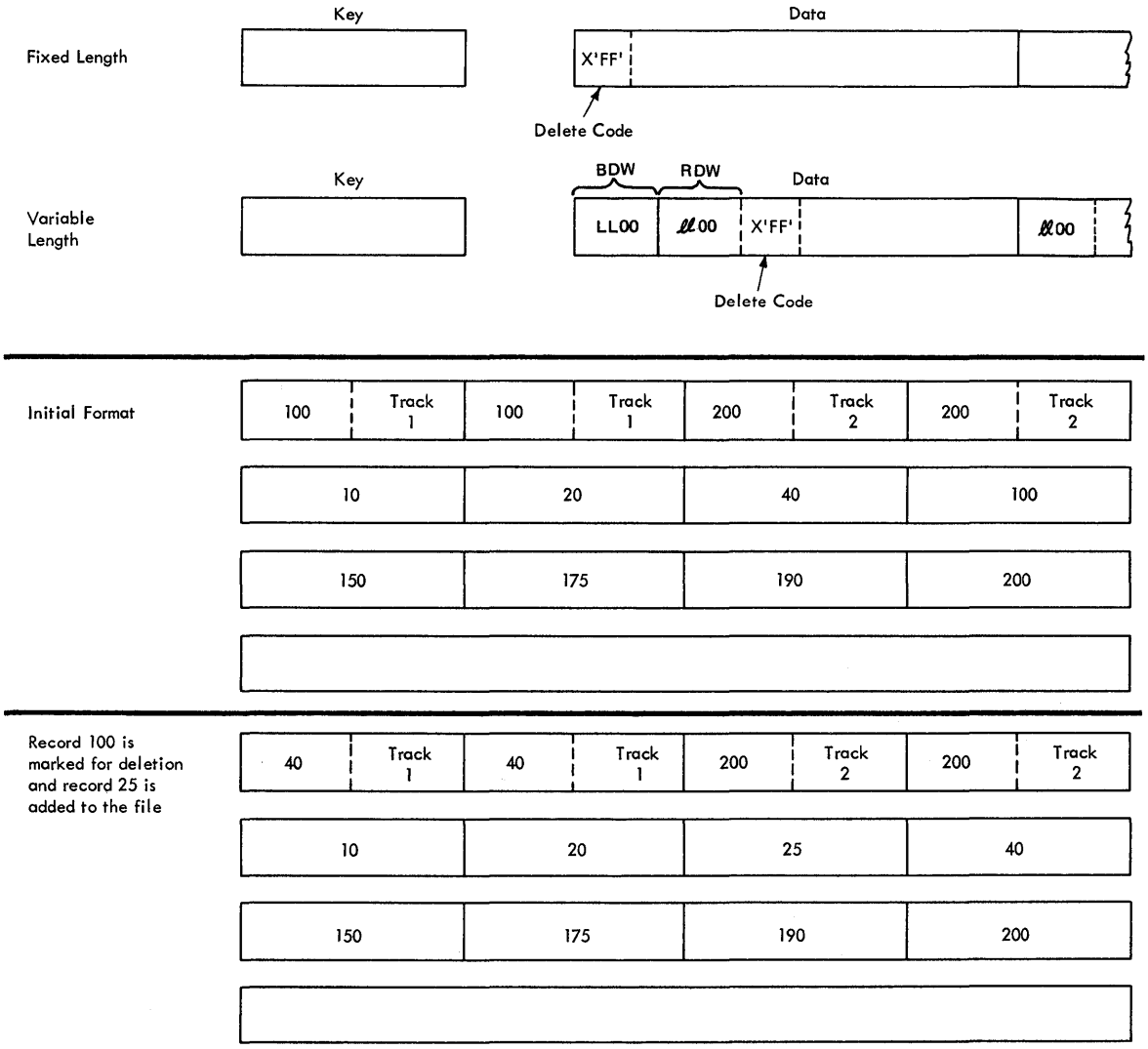


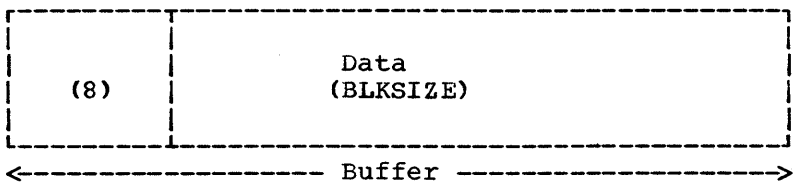
Figure 36. Deleting Records From an Indexed Sequential Data Set

If you indicate when creating the data set that you want to be able to flag records for deletion during updating, you can set the delete code to all ones (X'FF'). The delete code is the first byte of a fixed-length record or the fifth byte of a variable-length record. If a flagged record is forced off its prime track during a subsequent update, it will not be rewritten in the overflow area, as shown in Figure 36. Similarly, when you process sequentially, flagged records are not retrieved for processing. During direct processing, flagged records are retrieved like any other record and should be checked by you for the delete code.

INDEXED SEQUENTIAL BUFFER AND WORK AREA REQUIREMENTS

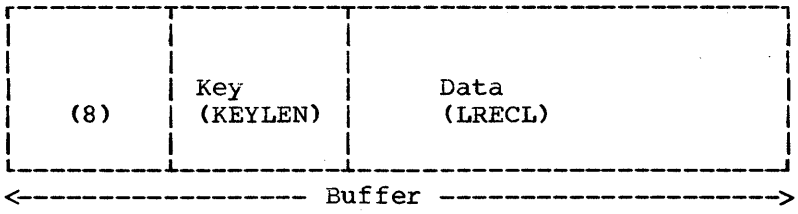
The only reason you will ever have to compute the buffer length (BUFL) requirements for your program is if you use the BUILD or GETPOOL macro instruction to construct the buffer area. If you are creating an indexed sequential data set (PUT macro instruction), each buffer must be eight bytes longer than the block size to allow for the hardware count field, that is:

Buffer length = 8 + Block size



One exception to this formula arises when dealing with unblocked format F records whose key field precedes the data field -- its relative key position is zero (RKP=0). In that case the key length must also be added, that is:

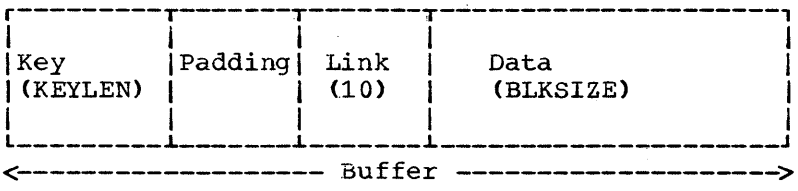
Buffer length = 8 + Key length + Record length



The buffer requirements for using the queued access method to read or update (GET or PUTX macro instruction) an indexed sequential data set are discussed below.

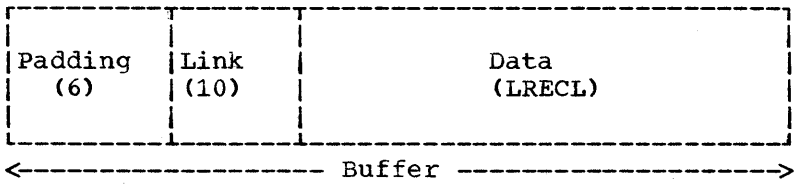
For fixed-length, unblocked records when both the key and data are to be read and for variable-length, unblocked records, padding is added so that the data will be on a doubleword boundary, that is:

Buffer length = Key length + Padding + 10 + Block size



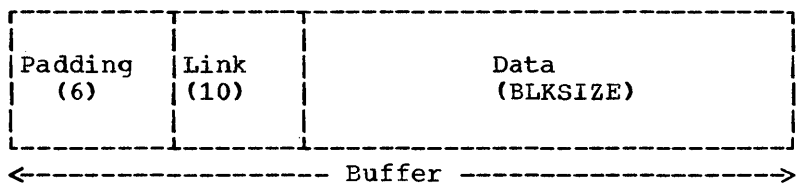
For fixed-length, unblocked records when only data is to be read:

Buffer length = 16 + LRECL



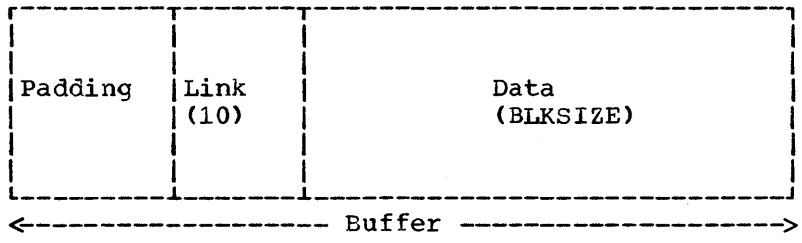
For fixed-length, blocked records:

Buffer length = 16 + BLKSIZE



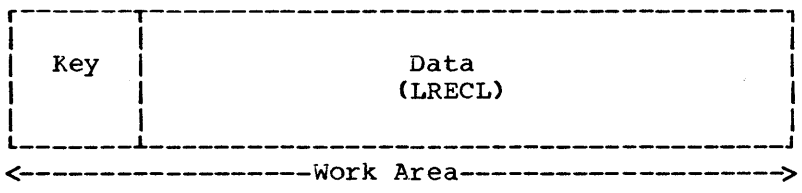
For variable-length, blocked records, padding is 2 if the buffer starts on a fullword boundary that is not also a doubleword boundary or it is 6 if the buffer starts on a doubleword boundary, that is:

Buffer length = 12 or 16 + Block size



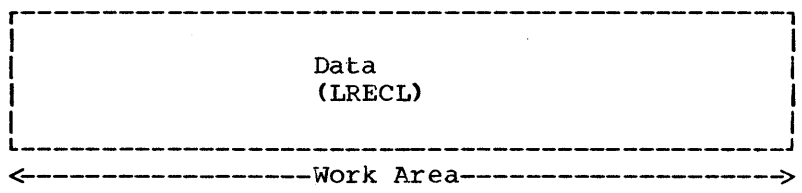
If you are using the input data set with fixed-length, unblocked records as a basis for creating a new data set, a work area is required. The size of the work area is given by:

Work area = Key length + Record length



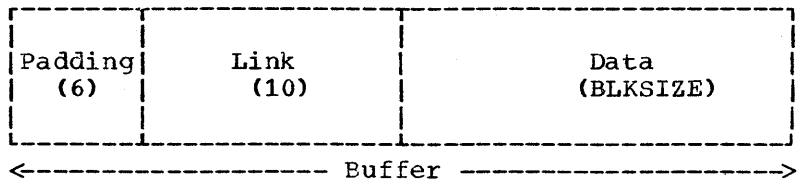
If you are reading only the data portion of fixed-length unblocked records or variable-length records, the work area is the same size as the record length, that is:

Work area = Record length



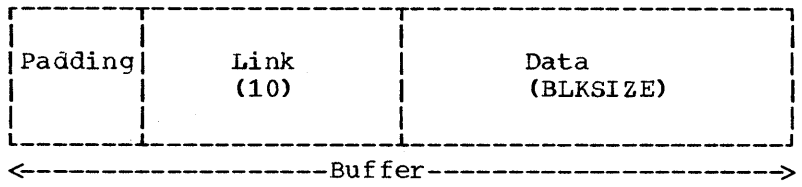
When using the basic access technique to update records in an indexed sequential data set, the key length field need not be considered in determining your buffer requirements. The area for fixed-length records must be:

Buffer length = 16 + Block size



For variable-length records, padding is 2 if the buffer starts on a fullword boundary that is not also a doubleword boundary or it is 6 if the buffer starts on a doubleword boundary. Thus, the area must be:

Buffer length = 12 or 16 + Block size



When adding variable-length records to a data set, you may provide a special work area for the operating system using the MSWA parameter of the DCB macro instruction. Although not required when adding fixed-length records, insertion is considerably expedited if you provide such an area. The size of the work area (SMSW parameter in the DCB) must be large enough to contain a full track of data plus the additional space to contain the count fields of each block and the work space for inserting the new record.

The size of the work area needed (SMSW parameter) varies according to the record format and the device type. You can calculate it during execution using device dependent information obtained with the DEVTYPE macro instruction and data set information from the data set control block (DSCB) obtained with the OBTAIN macro instruction. The DEVTYPE and OBTAIN macro instructions are discussed in the publication IBM System/360 Operating System: System Programmer's Guide.

Note that you can use the DEVTYPE macro instruction only if the index and prime areas are on the same type of device or if the index area is on a device with a larger track capacity than the device containing the prime area. If you are not trying to maintain device independence, you may precalculate the size of the work area needed and specify it in the SMSW field of the DCB macro instruction. The maximum value for SMSW is 65,535.

For calculating the size of the work area, refer to the storage device capacities shown in Table 16 under "Estimating Space Requirements" and the device overhead formulas given in the same section.

For fixed-length records, SMSW is calculated as follows:

$$SMSW = \left(\frac{\text{Track Capacity} - Bn}{Bi} + 1\right)(BLKSIZE + 8) + LRECL + KEYLEN$$

where Bn is the length of the last block on the track and Bi is the length of any block but the last as given in Table 17 in the section "Estimating Space Requirements".

For variable-length records, SMSW may be calculated by one of two methods. Method one may lead to faster processing although it may require more main storage than method two. For either method you must determine the value for HIRPD from the format 2 data set control block (DSCB). For the specific location of DS2HIRPD field in the data set control block, refer to the publication IBM System/360 Operating System: System Control Blocks.

Method one is as follows:

$$SMSW = HIRPD(BLKSIZE + 8) + LRECL + KEYLEN + 10$$

Method two is as follows:

$$SMSW = \left(\frac{\text{Track Capacity}-Bn}{Bi} + 2\right)(BLKSIZE) + 8(HIRPD) + LRECL + KEYLEN + 10$$

In all of the above formulas, the terms BLKSIZE, LRECL, KEYLEN, and SMSW are the same as the parameters in the DCB macro instruction. Method two yields a minimum value for SMSW. Therefore, method one is valid only if its application results in a value higher than one that would be derived from method two. If neither MSWA or SMSW are specified, the control program supplies the work area for variable-length records, using method two to calculate the size.

Another technique to increase the speed of processing is to provide space in main storage for the highest level index. To specify the address of this area, use the MSHI operand of the DCB. When the address of this area is specified, you must also specify its size, which you can do by using the SMSI operand of the DCB. The maximum value for SMSI is 65,535. If you do not use this technique, the index must be searched on the volume. The size of the storage area (SMSI parameter) varies. To allocate that space during execution, you can find the size of the index in the data control block (DCBNCRHI field) after the data set is opened or in the format 2 data set control block field, DS2NOBYT. Using the procedure discussed under the DCBD macro instruction, the storage area can be allocated and the SMSI field completed.

CONTROLLING AN INDEXED SEQUENTIAL DATA SET DEVICE

Processing of an indexed sequential data set is generally done in one of two ways: sequentially or directly. Direct processing is accomplished by using the basic access technique. Because you provide the key for the record you want read or written, all device control is handled automatically by the system. If you are processing the data set sequentially, using the queued access technique, the device is automatically positioned at the beginning of the data set.

In some cases, you may wish to process only a section or several separate sections of the data set. This is accomplished by using the SETL macro instruction, which directs the system to begin sequential retrieval at a specific record key. The processing of succeeding records is the same as for normal sequential processing, except that you must recognize when the last desired record has been processed. At this point, issue the ESETL macro instruction to terminate sequential processing. You can then begin processing at another point in the data set.

SETL -- Specify Start of Sequential Retrieval

The SETL macro instruction enables you to retrieve records starting at the beginning of an indexed sequential data set or at any point in the data set. Processing that is to start at a point other than the beginning can be requested in the form of a record key, a key prefix, or an actual address of a prime data record.

Use of a key prefix is extremely useful in that you do not have to know the whole key of the first record to be processed. Any number of key characters can be used in the key prefix. Key characters to the right should be represented by binary zeros.

In order to use actual addresses, you must keep an account of where the records were written when the data set was created. The device address of the block containing the record just processed by a PUT-move macro instruction is available in the 8-byte data control block field DCBLPDA. For blocked records the address is the same for each record in the block.

ESETL -- End Sequential Retrieval

The ESETL macro instruction directs the system to stop retrieving records from an indexed sequential data set. A new scan limit can then be set, or processing terminated. An end-of-data-set indication automatically terminates retrieval.

CREATING AN INDEXED SEQUENTIAL DATA SET

You can create an indexed sequential data set in one step or in several steps. You can create the data set either by writing all records in a single step or by writing one group of records in one step and writing additional groups of records in subsequent steps. Writing records in subsequent steps is resume loading. When using either one step or several steps, you must present the records for writing in ascending order by key.

To create an indexed sequential data set by the one-step method, you should proceed as follows:

- Code DSORG=IS or ISU and MACRF=PM or PL in the DCB macro instruction.

- Specify in the DD statement the DCB attributes DSORG=IS or ISU, record length (LRECL), block size (BLKSIZE), record format (RECFM), key length (KEYLEN), relative key position (RKP), options required (OPTCD), cylinder overflow (CYLOFL), and the number of tracks for a master index (NTM). Specify space requirements with the SPACE parameter. To reuse previously allocated space, omit the SPACE parameter and code DISP=(OLD,[KEEP]).

- Open the data set for output.

- Use the PUT macro instruction to place all the records or blocks on the direct access volume.

- Close the data set.

The records that comprise a newly created data set must be presented for writing in ascending order by key. You can merge two or more input data sets.

If records are blocked, you should not write a one-byte record with the hexadecimal value FF. This value is used for padding; if it occurs as the last record of a block, the record cannot be retrieved.

When creating an indexed sequential data set, a procedure called loading, you can increase performance by using the full track-index write option. You do this by specifying OPTCD=U in the DCB. This causes the operating system to accumulate track-index entries in main storage.

If you do not specify this option, the operating system writes each normal/overflow pair of entries for the track index after the associated prime data track has been written. If you specify this option, the operating system accumulates track-index entries in main storage until either there are enough entries to fill a track or end of data or end of cylinder is reached. Then the operating system writes these entries as a group, writing one group for each track of track index.

When you specify the full track-index write option, the track-index entries are written in the fixed, unblocked record format. If a large enough area of main storage is not available, the entries are written as they are created, that is, in normal/overflow pairs.

Once an indexed sequential data set has been created, its characteristics cannot be changed. However, for added flexibility, the system allows you to retrieve records using either the queued access technique with simple buffering, or the basic access technique with direct or dynamic buffering.

Tape-to-Disk -- Indexed Sequential Data Set: This example requires the creation of an indexed sequential data set from an input tape containing 60-character records. The key by which the data set is organized is in positions 20-29. The output records will be an exact image of the input, except that the records will be blocked. One track per cylinder is to be reserved for cylinder overflow. Master indexes are to be built when the cylinder index exceeds six tracks. Reorganization information about the status of the cylinder overflow areas is to be maintained by the system. The delete option will be used during any future updating.

```
//INDEXDD  DD     DSNAME=SLATE.DICT(PRIME),DCB=(BLKSIZE=240,CYLOFL=1,   C
//             DSORG=IS,OPTCD=MYLR,RECFM=FB,LRECL=60,NTM=6,RKP=19,      C
//             KEYLEN=10),UNIT=2311,SPACE=(CYL,25,,CONTIG),---
//INPUTDD  DD     ---
```

```
ISLOAD     START  0
           ...
           DCBD   DSORG=IS
ISLOAD     CSECT
           OPEN   (IPDATA,,ISDATA,(OUTPUT))
NEXTREC    GET    IPDATA            Locate Mode
           LR     0,1               Address of Record in Reg. 1
           PUT    ISDATA,(0)        Move Mode
           B      NEXTREC
           ...
CHECKERR   L      3,=A(ISDATA)      Initialize Base for Errors
           USING  IHADCB,3
           TM     DCBEXCD1,X'04'
           BO     OPERR             Uncorrectable Error
           TM     DCBEXCD1,X'20'
           BO     NOSPACE           Space Not Found
           TM     DCBEXCD2,X'80'
           BO     SEQCHK            Record Out of Sequence
*REST OF ERROR CHECKING
*ERROR ROUTINE
*END OF JOB ROUTINE (EODAD for IPDATA)
IPDATA     DCB    --
ISDATA     DCB    DDNAME=INDEXDD,DSORG=IS,MACRF=(PM),SYNAD=CHECKERR
```

To create an indexed sequential data set in more than one step, create the first group of records using the one step method described above. This first section must contain at least one data record. The remaining records can then be added to the end of the data set in subsequent steps using Resume Load. Each group to be added must contain records with successively higher keys. This method allows you to create the indexed sequential data set in several shorter time periods rather than in a single long one.

This method also allows you to provide limited recovery from uncorrectable output errors. When an uncorrectable output error is detected, do not attempt to continue processing or close the data set. If you have provided a SYNAD routine, it should issue the ABEND macro instruction to terminate processing. If no SYNAD routine is provided, the control program will terminate your processing. You should begin recovery at the record following the end of the data as of the last successful close. The rerun time is limited to that necessary only to add the new records, rather than to that necessary to recreate the whole data set.

When extending an indexed sequential data set with Resume Load, the disposition parameter of the DD statement must specify MOD. To insure that the necessary control information is in the data set control block before attempting to add records, you should at least open and close the data set successfully on a version of the system which includes Resume Load. This need be done only if the data set was created on a previous version of the system. Records may be added to the data set by this procedure until the space allocated for prime data in the first step has been filled.

During Resume Load on a data set with a partially filled track and/or a partially filled cylinder, the track index entry and/or the cylinder index entry is overlaid when the track or cylinder is filled. If Resume Load abnormally terminates after these index entries have been overlaid, a subsequent Resume Load will get a sequence check when

adding a key that is higher than the highest key at the last successful CLOSE but lower than the key in the overlaid index entry. When the SYNAD exit is taken for a sequence check, register 0 contains the address of the high key of the data set.

UPDATING AN INDEXED SEQUENTIAL DATA SET

In order to sequentially retrieve and update an indexed sequential data set:

- Code DSORG=IS or ISU, to agree with whichever one you specified when you created the data set, and MACRF=GL, SK, or PU in the DCB macro instruction.
- Code a DD statement for retrieving the data set. The data set characteristics and options are as defined when the data set was created.
- Open the data set for update.
- Set the beginning of sequential retrieval (SETL).
- Retrieve records and process as required marking nonoverflow records for deletion as required.
- Return records to the data set.
- End sequential retrieval as required and reset starting point (ESETL).
- Close the data set to end all retrieval.

Sequential Updates -- Indexed Sequential Data Set: Using the data set created in the previous example, you are to retrieve all records beginning with 915. Those records with a date (positions 13-16) previous to today's date are to be deleted. The date is in the standard form as returned by the system in response to the TIME macro instruction, i.e., packed decimal 00yyddds. If the record is an overflow record, the delete code is not to be entered.

```
-------------------------------------------------------------------
//INDEXDD  DD     DSNAME=SLATE.DICT,---
-------------------------------------------------------------------
ISRETR     START  0
           DCBD   DSORG=IS
ISRETR     CSECT
           ...
           USING  IHADCB,3
           LA     3,ISDATA
           OPEN   (ISDATA)
           SETL   ISDATA,KC,KEYADDR Set Scan Limit
           TIME                    Today's Date in Reg. 1
           ST     1,TODAY
NEXTREC    GET    ISDATA           Locate Mode
           CLC    19(10,1),LIMIT
           BNL    ENDJOB
           TM     DCBEXCD2,X'10'   Test for Overflow Record
           BO     NEXTREC
           CP     12(4,1),TODAY    Compare for Old Date
           BNL    NEXTREC
           MVI    0(1),X'FF'       Flag Old Record for Deletion
           PUTX   ISDATA           Return Delete Record
           B      NEXTREC
TODAY      DS     F
KEYADDR    DC     C'915'           Key Prefix
           DC     XL7'0'           Key Padding
LIMIT      DC     C'916'
           DC     XL7'0'
           ...
CHECKERR
*Test  DCBEXCD1 and DCBEXCD2 for error indication
*Error Routines
ENDJOB     CLOSE  (ISDATA)
           ...
ISDATA     DCB    DDNAME=INDEXDD,DSORG=IS,MACRF=(GL,SK,PU),          C
               SYNAD=CHECKERR
```

DIRECT RETRIEVAL AND UPDATE OF AN INDEXED SEQUENTIAL DATA SET

By using the basic access technique (BISAM) to process an indexed sequential data set, you can make direct references to the records in the data set for the purpose of:

- Direct retrieval of a record by its key.
- Direct update of a record.
- Direct insertion of new records.

Because the operations are direct, there can be no anticipatory buffering. However, the system provides a dynamic buffering service each time a read request is made, if specified.

To ensure that the requested record is in main storage before you start processing, you must issue a WAIT or CHECK macro instruction. If you issue a WAIT macro instruction, you must test the exception code field of the data event control block (DECB). If you issue a CHECK macro instruction, the system tests the exception code field in the data event control block (DECB). If an error analysis routine has not been specified and a CHECK is issued, the program will be abnormally terminated with a completion code X'001'. In either case, to determine whether the record is an overflow record, you should test the exception code field of the DECB.

After you test the exception code field of the DECB, you need not zero out this field. If you have used a READ KU macro instruction and

if you plan to use the same DECB again to rewrite the updated record using a WRITE K macro instruction, you should not zero out this field. If you do, your record may not be rewritten properly.

To update existing records, it is recommended that you use the READ, type KU, and WRITE, type K, combination. However, if you use a WRITE, type K, with a DECB not previously used to read the record, you are responsible for setting the overflow-record bit in the exception code field of the DECB. For blocked records, the overflow-record bit must be off when you write a prime block and on when you write an overflow block.

If there is a possibility that another program will require the use of the data set you are updating, you should ensure that you maintain exclusive control of at least the track. If you fail to maintain exclusive control of the data set that you are updating and if another data control block is opened before your data control block is closed, your updated records can become permanently inaccessible. In other words, when more than one data control block is open for updating a data set, the results are unpredictable. Exclusive control can be requested by using the ENQ macro instruction, which is described under "Supervisor Services."

Direct Update With Exclusive Control -- Indexed Sequential Data Set: In this problem the previously described data set is to be updated directly with transaction records on tape. The input tape records are 30 characters long; the key is in positions 1-10; the update information is in positions 11-30. The update information replaces data in positions 31-50 of the indexed sequential data record.

Exclusive control of the data set is requested since more than one task may be referring to the data set at the same time. Notice that exclusive control is released after each block is written to avoid tying up the data set until the update is completed.

```
//INDEXDD  DD      DSNAME=SLATE.DICT,DCB=(DSORG=IS,BUFNO=1,...),---
//TAPEDD   DD      ---
ISUPDATE  START   0
          ...
NEXTREC   GET     TPDATA,KEY
          ENQ     (RESOURCE,ELEMENT,E,,SYSTEM)
          READ    DECBRW,KU,MF=E
          WAIT    ECB=DECBRW
          TM      DECBRW+24,X'FD'      Test for any condition
          BM      RDCHECK              but overflow
          L       3,DECBRW+16          Pick up pointer to record
          MVC     30(20,3),UPDATE      Update record
          WRITE   DECBRW,K,MF=E
          WAIT    ECB=DECBRW
          TM      DECBRW+24,X'FD'      Any errors?
          BM      WRCHECK
          DEQ     (RESOURCE,ELEMENT,,SYSTEM)
          B       NEXTREC
RDCHECK   TM      DECBRW+24,X'80'      No record found
          BZ      SYNAD                If not, go to error routine
          FREEDBUF DECBRW,K,ISDATA     Otherwise, free buffer
          MVC     AREA,KEY
          WRITE   DECBRW,KN,,AREA-16,'S',MF=E   Add record to file
          WAIT    ECB=DECBRW
          TM      DECBRW+24,X'FD'      Test for errors
          BM      SYNAD
          DEQ     (RESOURCE,ELEMENT,,SYSTEM)   Release exclusive control
          B       NEXTREC
          DS      4F                   BISAM WRITE KN work field
AREA      DS      30C                  Logical record to be added
KEY       DS      CL10
UPDATE    DS      CL20
RESOURCE  DC      CL8'SLATE'
ELEMENT   DC      C'DICT'
          READ    DECBRW,KU,ISDATA,'S','S',KEY,MF=L
ISDATA    DCB     DDNAME=INDEXDD,DSORG=IS,MACRF=(RUS,WUA),              C
              MSHI=INDEX,SMSI=2000
TPDATA    DCB     ---
INDEX     DS      2000C
```

Note the use of the FREEDBUF macro instruction in the above example. Usually the FREEDBUF macro instruction has two functions:

- To indicate to the ISAM routines that a record that has been read for update will not be written back, and
- To free a dynamically obtained buffer.

In the above example, since the read operation was unsuccessful, the FREEDBUF macro instruction only frees the dynamically obtained buffer.

The first function of the FREEDBUF macro instruction described here allows you to read a record for update and then decide not to update it without performing a WRITE for update. You can use this function even when your READ macro instruction does not specify dynamic buffering, provided that you have included S (for dynamic buffering) in the MACRF field of your READ DCB.

However, you can accomplish an automatic FREEDBUF simply by reusing the DECB, that is, by issuing another READ or a WRITE KN to the same DECB. You should use this feature whenever possible, since it performs the functions of the FREEDBUF more efficiently. In the above

example, the FREEDBUF macro instruction should have been eliminated, since the WRITE KN addressed the same DECB as the READ KU.

For an indexed sequential data set with variable-length records, you may make three types of updates by using the basic access technique. You may read a record and write it back with no change in its length, simply updating some part of the record. This is done with a READ KU followed by a WRITE K, the same way you update fixed-length records. Two other methods for updating variable-length records use the WRITE KN macro instruction and allow you to change the record length. In one method, a record read for update (READ KU) may be updated in a manner which will change the record length and then be written back with its new length using WRITE KN. In the second method, you may replace a record with another record having the same key and possibly a different length using the WRITE KN macro instruction. To replace a record it is not necessary to have first read the record. In either method, when changing the record length, you must place the new length in the DECBLGTH field of the data event control block before issuing the WRITE KN macro instruction.

Direct Update -- Indexed Sequential Data Set with Variable-Length Records: In the following example an indexed sequential data set with variable-length records is updated directly with transaction records on tape. The transaction records are also of variable-length and each contains a code identifying the type of transaction. The transaction code 1 indicates that an existing record is to be replaced by one with the same key; 2 indicates that the record is to be updated by appending additional information, thus changing the record length; 3 or greater indicates that the record is to be updated with no change to its length. For this example, the maximum record length of both data sets is 256 bytes. The key is in positions 6-15 of the records in both data sets. The transaction code is in position 5 of records on the transaction tape. The work area (REPLAREA) is equal to the maximum record length plus 16 bytes.

```
//INDEXDD  DD    DSNAME=SLATE.DICT,DCB=(DSORG=IS,BUFNO=1,...),---
//TAPEDD   DD    ---
ISUPDVLR  START 0
          ...
NEXTREC   GET   TPDATA,TRANAREA
          CLI   TRANCODE,2                  Determine if replace or other
          BL    REPLACE                     Branch if replacement
          READ  DECBRW,KU,,'S','S',MF=E     Read record for update
          CHECK DECBRW,DSORG=IS             Check exceptional conditions
          CLI   TRANCODE,2                  Determine if change or
                                               append
          BH    CHANGE                      Branch if change
          ...
          ...
* CODE TO MOVE RECORD INTO REPLAREA+16 AND APPEND DATA FROM TRANSACTION
* RECORD
          ...
          MVC   DECBRW+6(2),REPLAREA+16     Move new length from RDW
          ...                               into DECBLGTH (DECB+6)
          WRITE DECBRW,KN,,REPLAREA,MF=E    Rewrite record with changed
          ...                               length
          CHECK DECBRW,DSORG=IS
          B     NEXTREC
CHANGE    ...
          ...
* CODE TO CHANGE FIELDS OR UPDATE FIELDS OF THE RECORD
          ...
          WRITE DECBRW,K,MF=E               Rewrite record with no
          ...                               change of length
          CHECK DECBRW,DSORG=IS
          B     NEXTREC
REPLACE   MVC   DECBRW+6(2),TRANAREA        Move new length from RDW
          ...                               into DECBLGTH (DECB+6)
          WRITE DECBRW,KN,,TRANAREA-16,MF=E Write transaction record
          ...                               as replacement for record
          ...                               with the same key
          CHECK DECBRW,DSORG=IS
          B     NEXTREC
CHECKERR  ...                               SYNAD Routine
          ...
REPLAREA  DS    CL272
TRANAREA  DS    CL4
TRANCODE  DS    CL1
KEY       DS    CL10
TRANDATA  DS    CL241
          READ  DECBRW,KU,ISDATA,'S','S',KEY,MF=L
ISDATA    DCB   DDNAME=INDEXDD,DSORG=IS,MACRF=(RUS,WUA),SYNAD=CHECKERR
TPDATA    DCB   ---
```

Processing a Direct Data Set

In a direct data set, there is a definite relationship between the control number or identification of each record and its location on the direct access volume. This relationship allows you to gain access to a record without an index search. The actual organization of the data set is completely determined by you. If the data set has been carefully organized, location of a particular record takes less time than with an indexed sequential data set.

A direct data set can only be processed by the basic access technique. For that reason, each unit of data transmitted between main storage and an I/O device is regarded by the system as a record.

If, in fact, it is a block, you must perform any blocking or deblocking required. For that reason, the BLKSIZE value must be equal to the LRECL value when format F or U records are processed. When format V records are used, the BLKSIZE value must be equal to the LRECL value plus four. Only BLKSIZE must be specified when adding or updating records on a direct data set.

As indicated in the discussion of direct access devices, record keys are optional. If they are specified, they must be used for every record and must be of a fixed length.

ORGANIZING A DIRECT DATA SET

In developing the organization of your data set, you can use a technique known as direct addressing. When direct addresses are used, the location of each record in the data set is known.

If format F records with keys are being written, the key of each record can be used. For example, a data set with keys ranging from 0 to 4999 should be allocated space of 5000 records. Each key relates directly to a location that you can refer to as a relative record number. The main disadvantage of this type of organization is that records may not exist for many of the keys even though space has been reserved for them.

Space could be allocated on the basis of the number of records in the data set rather than on the range of keys. This type of organization requires the use of a cross-reference table. When a record is written on the data set, you must note the physical location either as an actual address or as a relative track and record number. The addresses must then be stored in a table that is searched when a record is to be retrieved. Obvious disadvantages are that cross-referencing can only be used efficiently with a small data set; storage is required for the table; processing time is required for searching and updating the table.

A more common, but somewhat complex, technique for organizing the data set involves the use of indirect addressing. In indirect addressing, the address of each record in the data set is determined by a mathematical manipulation of the key. This manipulation is referred to as randomizing or conversion. Since a number of randomizing procedures could be used, no attempt is made here to describe or explain those that might be most appropriate for your data set.

REFERRING TO A RECORD IN A DIRECT DATA SET

Once you have determined how your data set is to be organized, you must consider how the individual records will be referred to when the data set is updated or new records are added. This is important for determining whether feedback will be required when creating the data; if so, in what form the returned address will be used. The record identification can be represented in any of the forms described below.

Relative Block Address: You specify the relative location of the record (block) within the data set as a 3-byte binary number. This type of reference can be used only with format F records. The system computes the actual track and record number.

Relative Track Address: You specify the relative track as a 2-byte binary number and the actual record number on that track as a 1-byte binary number.

Relative Track Address and Actual Key: In addition to the relative track address, you specify the address of a main storage location containing the record key. The system computes the actual track address and searches for the record with the correct key.

Actual Address: You supply the actual address in the standard 8-byte form -- MBBCCHHR. Remember, the use of an actual address may force you to indicate that the data set is unmovable.

Extended Search Option: You request that the system begin its search with a specified starting location and continue for a certain number of records or tracks. This same option can be used to request a search for unused space in which a record can be added.

To use the extended search option, you must indicate in the data control block the number of tracks (including the starting track) or records (including the starting record) that are to be searched. If you indicate a number of records, however, the system may actually examine more than this number. In searching a track, the system searches the whole track (starting with the first record); it therefore may examine records that precede the starting record or follow the ending record.

If the data control block specifies a number equal to or greater than the number of tracks allocated to the data set or the number of records within the data set, the entire data set is searched in the attempt to satisfy your request.

Exclusive Control for Updating: If more than one task in the same job step is referring to the same data set through the same data control block, exclusive control can be requested in the DCB macro instruction to prevent simultaneous reference to the same record. No other task requesting exclusive control of that record is given access to it until it is released by means of a WRITE or RELEX macro instruction.

CREATING A DIRECT DATA SET

Once the organization of a direct data set has been determined, the process of creating it is almost identical to that of creating a sequential data set. The data set organization field in the DCB macro instruction is specified as physical sequential (DSORG=PS or PSU). However, the DD statement must indicate direct access (DSORG=DA or DAU). The DCB macro instruction must specify a direct access device (DEVD=DA). If keys are used, a key length (KEYLEN) must also be specified. Record length (LRECL) should not be specified. The macro instruction form should indicate the WRITE macro instruction used to create a direct data set (WL).

If you are using a direct addressing technique with keys, you can reserve space for future records by writing a dummy record. A track for format U or V records can be reserved or truncated by writing a "capacity" record (see "Direct Access Device Characteristics").

Format F records are written sequentially as they are presented. When a track is filled, the system automatically writes the capacity record and advances to the next track. Because of the form in which relative track addresses are recorded, direct data sets to be accessed by means other than actual address must be limited in size to no more than 65,539 tracks for the entire data set.

Tape-to-Disk -- Direct Data Set: In this problem, a tape containing 204 character records arranged in key sequence is used to create a direct data set. A 4-byte binary key for each record ranges from 1000 to 8999, so space for 8000 records is requested.

```
//DAOUTPUT DD     DSNAME=SLATE.INDEX.WORDS,DCB=(DSORG=DA,                C
//              BLKSIZE=200,KEYLEN=4,RECFM=F),SPACE=(204,8000),---
//TAPINPUT DD     ---
```

```
DIRECT     START
           ...
           L       9,=F'1000'
           OPEN    (DALOAD,(OUTPUT),TAPEDCB)
           LA      10,COMPARE
NEXTREC    GET     TAPEDCB
           LR      2,1
COMPARE    C       9,0(2)                  Compare key of input against  C
               control number
           BNE     DUMMY
           WRITE   DECB1,SF,DALOAD,(2)     Write data record
           CHECK   DECB1
           AH      9,=H'1'
           B       NEXTREC
DUMMY      C       9,=F'8999'              Have 8000 records been written?
           BH      ENDJOB
           WRITE   DECB2,SD,DALOAD,DUMAREA  Write dummy
           CHECK   DECB2
           AH      9,=H'1'
           BR      10
INPUTEND   LA      10,DUMMY
           BR      10
ENDJOB     CLOSE   (TAPEDCB,,DALOAD)
           ...
DUMAREA    DS      CL5
DALOAD     DCB     DSORG=PS,MACRF=(WL),DDNAME=DAOUTPUT,                   C
               DEVD=DA,SYNAD=CHECKER,---
TAPEDCB    DCB     EODAD=INPUTEND,MACRF=(GL), ---
```

ADDING/UPDATING RECORDS ON A DIRECT DATA SET

The facilities and the techniques for adding records to a direct data set depend to a great extent on the format of the records and the organization used.

Format F With Keys: The add function is essentially an update by record identification. The reference to the record can be made by either a relative block address or a relative track address.

If you attempt to add a record by relative block address, the system converts the address to a relative track. That track is searched and the new record written in place of the first "dummy" record on the track. If there is no dummy record on the track, you are informed that the write operation did not take place. However, if you request the extended search option, the new record will be written in place of the first dummy record found within the search limit you specify. If none is found, you are notified that the write operation could not take place. In the same way, a reference by relative track address causes the record to be written in place of the first dummy record on that track or the first within the search limit, if requested.

Format F Without Keys: Here too, the add function is really an update of dummy records already in the data set. The main difference is that dummy records cannot be written automatically when the data set is created. You will have to use your own method for flagging dummy records. The update form of the WRITE macro instruction (MACRF=W) must be used rather than the add form (MACRF=WA).

You will have to retrieve the record first (READ macro instruction), test for a dummy record, update, and write.

Format V or U With Keys: The technique used to add records in this case depends on the way the data set is organized -- indirect addressing or cross-reference table. If indirect addressing is used to create the data set, you must at least initialize each track (write a capacity record) even if no data is actually written. That way the capacity record indicates how much space is available on the track.

If a cross-reference table is used, you should exhaust the input and then initialize enough succeeding tracks to contain any additions that might be required.

To add a new record, use a relative track address. The system examines the capacity record to see if there is room on the track. If there is, the new record is written. Under the extended search option, the record is written in the first available area within the search limit.

Format V or U Without Keys: This format does not lend itself to making additions. You can refer to a record only by its relative track or actual device address.

Tape-to-Disk Add -- Direct Data Set: This problem involves adding records to the data set created in the last example. Notice that the write operation adds the key and the data record to the data set. If the existing record is not a dummy record, an indication is returned in the exception code of the DECB. For that reason, it is better to use the WAIT macro instruction instead of the CHECK macro instruction to test for errors or exceptional conditions.

```
//DIRADD   DD     DSNAME=SLATE.INDEX.WORDS,---
//TAPEDD   DD     ---
```

```
DIRECTAD  START
          ...
          OPEN    (DIRECT,(OUTPUT),TAPEIN)
NEXTREC   GET     TAPEIN,KEY
          L       4,KEY                 Set up relative record number
          SH      4,=H'1000'
          ST      4,REF
          WRITE   DECB,DA,DIRECT,DATA,'S',KEY,REF+1
          WAIT    ECB=DECB
          CLC     DECB+1(2),=X'0000'  Check for any errors
          BE      NEXTREC
* Check error bits and take required action
DIRECT    DCB     DDNAME=DIRADD,DSORG=DA,RECFM=F,KEYLEN=4,BLKSIZE=200, C
              MACRF=(WA)
TAPEIN    DCB     ---
KEY       DS      F
DATA      DS      CL200
REF       DS      F
          ...
```

Tape-to-Disk Update -- Direct Data Set: This problem is similar to the previous example. However, since you are updating, there is no check for dummy records. The existing direct data set contains 25,000 records whose 5-byte keys range from 00001 to 25,000. Each data record is 100 bytes long. The first 30 characters are to be updated.

The input tape records are 35 characters long -- 5-byte key and 30-byte data. Notice that only data is brought into main storage for updating.

```
-------------------------------------------------------------------------
//DIRECTDD DD     DSNAME=SLATE.INDEX.WORDS,---
//TAPINPUT DD     ---
-------------------------------------------------------------------------
DIRUPDAT   START
           ...
           OPEN   (DIRECT,(UPDAT),TAPEDCB)
NEXTREC    GET    TAPEDCB,KEY
           PACK   KEY,KEY
           CVB    3,KEYFIELD
           SH     3,=H'1'
           ST     3,REF
           READ   DECBRD,DI,DIRECT,'S','S',0,REF+1
           CHECK  DECBRD
           L      3,DECBRD+12
           MVC    0(30,3),DATA
           ST     3,DECBWR+12
           WRITE  DECBWR,DI,DIRECT,'S','S',0,REF+1
           CHECK  DECBWR
           B      NEXTREC
           ...
KEYFIELD   DS     0D
           DC     XL3'0'
KEY        DS     CL5
DATA       DS     CL30
REF        DS     F
DIRECT     DCB    DSORG=DA,DDNAME=DIRECTDD,MACRF=(RISC,WIC),          C
               OPTCD=R,BUFNO=1
TAPEDCB    DCB    ---
           ...
```

Consideration for User Labels: User labels must be created when the data set is created. They may be updated when processing a direct data set but not added or deleted. When creating a multi-volume direct data set using BSAM, you should turn off the header exit entry after OPEN and turn on the trailer label exit entry just prior to issuing the CLOSE. This eliminates the end-of-volume exits. The first volume, containing the user label track, must be mounted at CLOSE time. If you have requested exclusive control, OPEN/CLOSE will use ENQ/DEQ facilities to prevent simultaneous reference to user labels.

Part 3: Data Set Disposition and Space Allocation

Allocating Space on Direct Access Volumes

When direct access storage space is required for a data set, you have to specify the amount of space needed and the device type. The operating system selects the device and allocates the space accordingly. This facility provides for more flexible and efficient use of devices and available storage space. It also relieves you of the responsibility and details involved in efficient space control.

Before a direct access volume can be used for data storage, it must be initialized by the utility program, Direct Access Storage Device Initialization (DASDI). The DASDI functions include in part:

- Creating the standard 80-byte volume label and writing it on cylinder 0, track 0, of the volume.
- Initializing the volume table of contents (VTOC). The location of the VTOC depends upon the conventions used by your installation when initializing the volume.
- Writing the home address (HA) and capacity record (R0) for each track.
- Checking tracks and making alternate track assignments if necessary.

When the data set is to be stored on a direct access volume, you must supply control information designating the amount of space to be allocated and in what manner. This information is supplied in the data definition (DD) statement for the data set.

SPECIFYING SPACE REQUIREMENTS

The amount of space required can be specified in terms of blocks, tracks, or cylinders. If you want to maintain device-independence across direct access device types, specify your space requirements in terms of blocks. Otherwise, if your request is in terms of tracks or cylinders, you must be aware of such device considerations as cylinder or track capacity.

Cylinder allocation allows faster input/output of sequential data sets than does track allocation. Track allocation stops input/output at the end of every track to prevent references on the same cylinder outside of the data set. This difference occurs only when reading, and then only when the records are not in the fixed standard (FS) format.

Allocation by Blocks: When the amount of space required is expressed in terms of blocks, you must specify the number and average block length of the blocks within the data set, e.g.:

```
//  DD   --,SPACE=(300,(5000,100))

     300 = average block length.
    5000 = quantity (number of blocks).
     100 = increment (to be used if the quantity is not sufficient)
           allocated in terms of additional blocks.
```

Note: When average block and secondary space allocation are being used, the BLKSIZE parameter specified must be equal to the maximum block length.

From this information, the operating system estimates and allocates the number of tracks required. Space is always allocated in whole track units. You may also request that the space allocated for a specific number of blocks begin and end on cylinder boundaries.

You must be certain that both the quantity and increment are large enough to contain the largest block to be written. Otherwise, all of the space requested is allocated but erased as the system tries to find a space large enough for the record.

Allocation by Tracks or Cylinders: When the amount of space required is expressed in terms of tracks or cylinders, you must also specify the device type in the DD statement, e.g.:

```
//  DD   --,SPACE=(TRK,(100,5)),UNIT=2301
//  DD   --,SPACE=(CYL,(3,1)),UNIT=2311
```

Allocation by Absolute Address: If the data set contains location-dependent information in the form of an absolute track address, i.e., MBBCCHHR, space should be requested in terms of the number of tracks and the beginning address, e.g.:

```
//  DD   --,SPACE=(ABSTR,(500,20)),UNIT=2311
```

where: 500 tracks are required beginning at relative track 20.

Additional Space Allocation Options: The DD statement provides you with a great deal of flexibility in specifying space requirements. You can request that the space be contiguous (CONTIG) or separated (SPLIT). These and other options are described in detail in the publication IBM System/360 Operating System: Job Control Language.

ESTIMATING SPACE REQUIREMENTS

In order to determine how much space your data set requires, you must consider a number of variables:

- Device type.
- Track capacity.
- Tracks per cylinder.
- Cylinders per volume.
- Data length (block size).
- Key length.
- Device overhead.

Table 16 lists the physical characteristics of a number of direct access storage devices.

Table 16. Direct Access Storage Device Capacities

Device Type	Volume Type	Track Capacity*	Tracks/Cylinder	No. of Cylinders	Total Capacity*
2311	Disk	3625	10	200	7,250,000
2314	Disk	7294	20	200	29,176,000
2302	Disk	4984	46	246	56,398,944
2303	Drum	4892	10	80	3,913,600
2301	Drum	20483	8	25**	4,096,600
2321	Cell	2000	20***	980***	39,200,000

*Capacity indicated in bytes.
**There are 25 logical cylinders in a 2301 Drum.
***A volume is equal to one bin in a 2321 Data Cell.

The term "device overhead" refers to the space required on each track for hardware data, i.e., address markers, count areas, gaps between records, R0, etc. Device overhead varies with each device and depends also on whether the blocks are written with keys. To compute the actual space required for each block including device overhead, you can use the formulas in Table 17.

Table 17. Direct Access Device Overhead Formulas

Device	Bytes Required by Each Data Block			
	Blocks With Keys		Blocks Without Keys	
	Bi	Bn	Bi	Bn
2311	81+1.049(KL+DL)	20+KL+DL	61+1.049(DL)	DL
2314	146+1.043(KL+DL)	45+KL+DL	101+1.043(DL)	DL
2302	81+1.049(KL+DL)	20+KL+DL	61+1.049(DL)	DL
2303	146+KL+DL	38+KL+DL	108+DL	DL
2301	186+KL+DL	53+KL+DL	133+DL	DL
2321	100+1.049(KL+DL)	16+KL+DL	84+1.049(DL)	DL

Bi is any block but the last on the track.
Bn is the last block on the track.
DL is data length.
KL is key length.

The formulas can be combined in the following way:

If you intend to specify your space requirements in terms of tracks (TRK) or cylinders (CYL), your estimate should be made as shown above. If you request absolute tracks (ABSTR), remember that you cannot allocate track 0, cylinder 0. The amount of space required for the volume table of contents will reduce the space available on the rest of the volume.

On the other hand, if you specify your space requirements in terms of average block length, the system performs the computations for you.

Because a sequential data set and a direct data set are created in the same way, the estimate and specification of space requirements are identical. If you use the WRITE SZ macro instruction, your secondary allocation for a direct data set should be at least two tracks. Space allocation for a partitioned data set requires that you also consider the space used for the directory. Similarly, allocation for an

indexed sequential data set requires that you consider the space needed for the prime area, index areas, and overflow areas.

ALLOCATING SPACE FOR A PARTITIONED DATA SET

What is the average size of the members to be stored on your direct access volume? How many members will fit on the volume? Will you need directory entries for the member names only or will aliases be used? How many? Will members be added or replaced frequently? All of these questions must be answered if you are to estimate your space requirements accurately and use the space efficiently. Note, too, that a partitioned data set cannot extend beyond one volume.

If your data set will be quite large, or you expect to do a lot of updating, it might be best to allocate a full volume. If it will be small or seldom subject to change, you should make your estimate as accurate as possible to avoid wasted space or wasted time used for recreating the data set.

Because the characteristics of all the members of the data set must be uniform, the record format could be specified as undefined (RECFM=U) and the block size (BLKSIZE) as a maximum length. It is a good practice to indicate a block length equal to track capacity, e.g., BLKSIZE=3625 for a 2311 disk. You might then ask for either 200 tracks, or 20 cylinders, thus allowing for 725,000 bytes of data.

Assuming an average length of 70,000 bytes for each member, you need space for at least 10 directory entries. If each member also has an average of three aliases, space for an additional 30 directory entries is required.

Space for the directory is expressed in terms of 256-byte blocks. Each block contains from three to twenty entries, depending on the length of the user data field. If you expect 40 directory entries, request at least eight blocks. Because the space for the directory is allocated in full track units, any unused space on the track is wasted unless there is enough space left to contain a block of the first member. Therefore, the most advisable request in this case would be for 10 blocks.

Putting the space estimates into specification form, any of the following would cause the same allocation:

```
SPACE=(3625,(200,,10))
SPACE=(CYL,(20,,10))
SPACE=(TRK,(200,,10))
```

Although an increment has been omitted in these examples, it could have been supplied to provide for extension of the member area. The directory size, however, cannot be extended.

ALLOCATING SPACE FOR AN INDEXED SEQUENTIAL DATA SET

An indexed sequential data set can be divided into three areas: prime, index, and overflow. Space for these areas can be subdivided and allocated in several different ways:

- Prime area -- If you request space in terms of a prime area only, the system automatically uses a portion of that space for indexes, taking one cylinder at a time as needed. Any unused space in the last cylinder used for index will be allocated as an independent overflow area. More than one volume can be used in most cases, but all volumes must be of the same device type.

- Index area -- You can request that a separate area be allocated to contain your cylinder and master indexes. The index area must be contained within one volume, but this volume need not be of the same device type as the prime area volume. If a separate index area is requested, you cannot catalog the data set with a DD statement.

  A slight variation for requesting an index area can be used if the total space occupied by the prime area and index area does not exceed one volume. In this case, you can request that the separate index area be embedded in the middle of the prime area (to reduce access arm movement) by indicating an index size in the SPACE parameter of the DD statement defining the prime area.

  If you request space in terms of prime and index areas only, the system will automatically use any space remaining on the last cylinder used for master and cylinder indexes for overflow, provided the index area is on the same type of device as the prime area.

- Overflow area -- Although you can request an independent overflow area, it must be contained within one volume. If no specific request for index area is made, then it will be allocated from the specified independent overflow area.

  To request that a designated number of tracks on each cylinder be used for cylinder overflow records, you must use the CYLOFL parameter of the DCB macro instruction. The number of tracks that you can use on each cylinder equals the total number of tracks on the cylinder minus the sum of the tracks needed for track index and the tracks required for prime data, that is:

  Useable tracks =

  total tracks - (track index tracks + prime data tracks)

Note that when you create a one-cylinder data set, ISAM reserves one track on the last cylinder for the end-of-file filemark.

When requesting space for an indexed sequential data set, the DD statement must follow a number of conventions, as shown below and summarized in Table 18.

• Table 18. Requests for Indexed Sequential Data Sets

Criteria			Restrictions on Unit Types and Number of Units Requested	Resulting Arrangement of Areas
1. Number of DD Statements	2. Types of DD Statements	3. Index Size Coded		
3	INDEX PRIME OVFLOW	-	None	Separate index, prime, and overflow areas.
2	INDEX PRIME	-	None	Separate index and prime areas.
2	PRIME OVFLOW	No	None	Prime area and overflow area with an index at its end.
2	PRIME OVFLOW	Yes	The statement defining the prime area cannot request more than one unit.	Prime area and embedded index, and overflow area.
1	PRIME	No	None	Prime area with index at its end. Any unused index area will be used for independent overflow.
1	PRIME	Yes	Statement cannot request more than one unit.	Prime area with embedded index area.

- Space (SPACE) can be requested only in terms of cylinders (CYL) or absolute tracks (ABSTR). If the absolute track technique is used, the designated tracks must encompass an integral number of cylinders.
- Data set organization (DSORG) must be specified as indexed sequential (IS or ISU) in both the DCB macro instruction and the DCB parameter of the DD statement.
- All required volumes must be mounted when the data set is opened, i.e., volume mounting cannot be deferred.
- If your prime area extends beyond one volume, you must indicate the number of units and volumes to be spanned, e.g., UNIT=(2311,3),VOLUME=(,,,3).
- You can catalog the data set using the DD statement parameter DISP=(,CATLG) only if the entire data set is defined by one DD statement, i.e., you did not request a separate index or independent overflow area.

As your data set is created, the operating system builds the track indexes in the prime data area. Unless you request a separate index area or an embedded index area, the cylinder and master indexes are built in the independent overflow area. If you did not request an independent overflow area, the cylinder and master indexes are built from the prime area.

Note: If an error is encountered when allocating a multivolume data set, the IEHPROGM utility program should be used to scratch the data set control blocks of the data sets that were successfully allocated. The IEHLIST utility program can be used to determine whether or not part of the data set has been allocated. The IEHLIST utility program is also useful to determine whether space is available or whether identically named data sets exist before space allocation is attempted for indexed sequential data sets. These utility programs are described in the publication IBM System/360 Operating System: Utilities.

SPECIFYING A PRIME DATA AREA

To request that the system allocate space and subdivide it as required, you should code:

```
//ddname   DD  DSNAME=dsname,DCB=DSORG=IS,                          C
//             SPACE=(CYL,quantity,,CONTIG),UNIT=unitname,          C
//             DISP=(,KEEP),---
```

You can accomplish the same type of allocation by qualifying your dsname with the element indication (PRIME). This element is assumed if omitted. It is required only if you request an independent index or overflow area. To request an embedded index area when an independent overflow area is specified, you must indicate DSNAME=dsname(PRIME). To indicate the size of the embedded index, you specify SPACE=(CYL, (quantity,,index size)).

SPECIFYING A SEPARATE INDEX AREA

In order to request a separate index area, other than an embedded area as described above, you must use a separate DD statement. The element name is specified as (INDEX). The space and unit designations are as required. Notice that only the first DD statement can have a data definition name. The data set name (dsname) must be the same.

```
//ddname   DD  DSNAME=dsname(INDEX),---
//         DD  DSNAME=dsname(PRIME),---
```

SPECIFYING AN INDEPENDENT OVERFLOW AREA

A request for an independent overflow area is essentially the same as for a separate index area. Only the element name, OVFLOW, is changed. If you do not request a separate index area, only two DD statements are required.

```
//ddname   DD  DSNAME=dsname(INDEX),---
//         DD  DSNAME=dsname(PRIME),---
//         DD  DSNAME=dsname(OVFLOW),---
```

CALCULATING SPACE REQUIREMENTS FOR AN INDEXED SEQUENTIAL DATA SET

To determine the number of cylinders required for an indexed sequential data set, you must consider the number of blocks that will

fit on a cylinder, the number of blocks that will be processed, and the amount of space required for indexes and overflow areas. In making the computations, consider additional space that is required for device overhead as shown in Table 16. Remember the formula:

$$\text{Blocks per track} = 1 + \frac{\text{Track capacity - Length of the last block}}{\text{Length of other blocks}}$$

$$Bt = 1 + ((Ct-Bn)/Bi)$$

Step 1

Once you know how many records will fit on a track and the maximum number of records you expect to create, you can determine how many tracks you will need for your data.

$$\text{Number of tracks required} = \frac{\text{Maximum number of blocks}}{\text{Blocks per track}} + 1$$

ISAM load mode reserves the last prime data track for the filemark.

Example: Assume the existence of a 200,000 record part-of-speech dictionary to be stored on an IBM 2311 Disk Storage Unit as an indexed sequential data set. Each record in the dictionary has a 12-byte key (the word itself) and an 8-byte data area containing a part-of-speech code and control information. Each block contains 50 records -- LRECL=20 and BLKSIZE=1000. Using the formula from Table 16, we find that each track will contain 3 blocks or 150 records. A total of 1333 1/3 tracks will be required for the dictionary.

$$Bt = 1 + \frac{3625 - (20 + 12 + 1000)}{81 + 1.049(12 + 1000)} = 1 + \frac{2593}{1143} = 3$$

Records per Track = (3 blocks)(50 records per block) = 150

$$\text{Prime data tracks required (T)} = \frac{200{,}000 \text{ records}}{150 \text{ records per track}} + 1 = 1334\ 1/3$$

Step 2

You will want to anticipate the number of tracks required for cylinder overflow areas. The computations formula is the same as for prime data tracks, but you must remember that overflow records are unblocked and a 10-byte link field is added. Remember, if you exceed the space allocated for any cylinder overflow area, an independent overflow area is required. Those records are not placed in another cylinder overflow area.

$$\text{Overflow records per track} = 1 + \frac{\text{Track capacity - Length of last overflow record}}{\text{Length of other overflow records}}$$

$$Ot = 1 + ((Ct-Rn)/Ri)$$

Example: Approximately 5000 overflow records are expected for the data set described in step 1. Since 29 overflow records will fit on a track, 173 overflow tracks are required. This is approximately 2 overflow tracks for every 15 prime data tracks. Since the 2311 disk has 10 tracks per cylinder, it would probably be best to allocate 2 tracks per cylinder for overflow.

$$Ot = 1 + \frac{3625 - (20 + 12 + 20 + 10)}{81 + 1.049(12 + 20 + 10)} = 1 + \frac{3563}{126} = 29$$

Overflow tracks required = $\frac{\text{5000 records}}{\text{29 records per track}} = 173$

Overflow tracks per cylinder (Oc) = 2

Step 3

You will have to set aside space in the prime area for track index entries. There will be two entries (normal and overflow) for each track on a cylinder that contains prime data records. The data field of each index entry is always 10 bytes. The key length corresponds to the key length for the prime data records. How many index entries will fit on a track?

Index entries per track = $1 + \frac{\text{Track capacity - Length of last index entry}}{\text{Length of other index entries}}$

It = 1 + ((Ct-En)/Ei)

Example: Again assuming a 2311 disk and records with a 12-byte key, we find that 35 index entries will fit on a track.

$$It = 1 + \frac{3625 - (20 + 12 + 10)}{81 + 1.049(12 + 10)} = 1 + \frac{3583}{105} = 1 + 34 = 35$$

Step 4

The number of tracks required for track index entries will depend on the number of tracks per cylinder and the number of track index entries per track. Any unused space on the last track of the track index can be shared with prime data records if they will fit.

Number of track index tracks per cylinder = $\frac{2(\text{Tracks per cylinder}) + 1}{\text{Index entries per track} + 2}$

Ic = (2Tc+1)/(It+2)

Note: For variable-length records, the last track of the track index is not shared with prime data records.

Example: The 2311 disk has 10 tracks per cylinder. You can fit 35 track index entries per track. Therefore, you need less than one track for each cylinder:

$$Ic = \frac{2(10) + 1}{35 + 2} = \frac{21}{37}$$

The space remaining on the track is ((1-21/37)(3625)) = 1567 bytes. This is enough for one block of prime data records. Since the normal number of blocks per track is 3, the block uses one third of the track, and the effective value of Ic is therefore 1-1/3 = 2/3.

Step 5

Next you have to compute the number of tracks available on each cylinder for prime data records. You cannot include tracks set aside for cylinder overflow records.

Prime data tracks per cylinder = Tracks per cylinder - Overflow tracks per cylinder - Index tracks per cylinder

$$Pc = Tc - Oc - Ic$$

Example: If you set aside 2 cylinder overflow tracks, and you require 2/3 of a track for the track index, 7 1/3 tracks are available on each cylinder for prime data records.

$Pc = 10 - 2 - 2/3 = 7\ 1/3$

Step 6

The number of cylinders required for the prime data records, track index area, and cylinder overflow area is determined by the number of prime data tracks required divided by the number of prime data tracks available on each cylinder.

Number of cylinders required = Prime data tracks required / Prime data tracks per cylinder

$$C = T/Pc$$

Example: You need 1333 1/3 tracks for prime data records. You can use 7 1/3 tracks per cylinder. Therefore, 182 cylinders are required for your prime area and cylinder overflow areas.

$$C = \frac{1333\ 1/3}{7\ 1/3} = 181.9$$

Step 7

You will need space for a cylinder index as well as track indexes. There is a cylinder index entry for each track index, i.e., for each cylinder allocated for the data set. The size of each entry is the same as the size of the track index entries; therefore, the number of entries that will fit on a track is the same as the number of track index entries. Unused space on a cylinder index track is not shared.

Number of tracks required for cylinder index = Track indexes + 1 / Index entries per track

$$Ci = (C+1)/It$$

Example: You have 182 track indexes. Since 35 index entries fit on a track, you need 5.3 tracks for your cylinder index. The remaining space on the last track is unused.

$$Ci = \frac{182 + 1}{35} = 5.3$$

Step 8

If you have a data set large enough to require master indexes, you will want to calculate the space required according to the number of tracks for master indexes (NTM parameter) you specified in the DCB macro instruction or the DD statement.

If the cylinder index exceeds the NTM specification, an entry is made in the master index for each track of the cylinder index. If the master index itself exceeds the NTM specification, a second level master index is started. Up to three levels of master indexes are created if required.

The space requirements for the master index are computed in the same way as the cylinder index.

Number of tracks required for master indexes = $\frac{\text{Cylinder index tracks}}{\text{Index entries per track}} + 1$

$M_1 = (Ci+1)/It$ when $Ci>NTM$

$M_2 = (M_1+1)/It$ when $M_1>NTM$

$M_3 = (M_2+1)/It$ when $M_2>NTM$

Example: Assume that your cylinder index will require 22 tracks. Since large keys are used, only 10 entries will fit on a track. Assuming that NTM was specified as 2, 3 tracks will be required for a master index, and two levels of master index will be created.

$M_1 = (22+1)/10 = 2.3$

Summary: Indexed Sequential Space Requirement Calculations

1. How many blocks will fit on a track?

   $Bt = 1 + ((Ct-Bn)/Bi)$

2. How many overflow records will fit on a track?

   $Ot = 1 + ((Ct-Rn)/Ri)$

3. How many index entries will fit on a track?

   $It = 1 + ((Ct-En)/Ei)$

4. How many track index tracks are needed per cylinder?

   $Ic = (2Tc+1)/(It+2)$

5. How many tracks on each cylinder can be used for prime data records?

   $Pc = Tc - Oc - Ic$

6. How many cylinders are needed for the prime data area?

   $C = \frac{T}{Pc}$

7. How many tracks are required for the cylinder index?

   $Ci = (C+1)/It$

8. How many tracks are required for master indexes?

   $M = (Ci+1)/It$

Control and Disposition of Data Sets

There are two levels of status and disposition of the data sets you use for your processing. The status and disposition information must be provided to the system in the disposition field of the DD statement, DISP=(status,disposition). The first level deals with the status of the data set when you begin processing and the relationship of the data set to other job steps in your job or other jobs. The second deals with what is to be done with the data set when you have completed processing. It is at this level of control and disposition that you can take advantage of the cataloging facilities of the operating system.

A data set that is being used for input has a status of OLD. If it can be used by more than one job, the status should be specified as SHR. If you are going to add to the input data set, specify MOD. The system automatically positions the access mechanism after the last record when the data set is opened. A NEW output data set should be so indicated.

Having identified the status of the data set at the beginning of your job step, you should specify how you want it disposed of at the end of processing. If the disposition is to be unchanged, you need not specifyanything more. The status of an existing data set remains unchanged; a new data set is deleted.

The requested disposition is performed at the end of the job step. A data set to be used in a later job can be kept (KEEP) until a subsequent request is made to DELETE it. If the data set is to be used by more than one job step in the same job, you can specify that it is to be passed (PASS).

The most useful disposition provided by the system is the cataloging facility (CATLG). The data set name is recorded by the system and its volume noted. An old data set can subsequently be removed from the catalog if you so request (UNCATLG).

If you wish, you can specify one disposition to be performed if the job step terminates normally, and a different disposition to be performed if the job step terminates abnormally. For example, you can specify DISP=(OLD,DELETE,KEEP) if you wish to delete a data set under normal conditions, but wish to keep it if processing is abnormally terminated. For normal termination, you can specify any disposition -- PASS, KEEP, DELETE, CATLG, or UNCATLG; for abnormal termination, you can specify any disposition except PASS.

ROUTING DATA SETS THROUGH THE OUTPUT STREAM

Data sets that are to be printed or punched can be routed through the output stream. This allows greater flexibility in scheduling print and punch operations, and improves operating system efficiency.

When you route a data set through the output stream, you do not request a unit record device for exclusive use by your job step. Instead, you assign the data set to an output class, which may include data sets from many different jobs. Output classes are identified by the letters A-Z and the digits 0-9. Each is associated with a specific device type. By convention, class A consists of high priority output to be printed (e.g., a listing of job control statements), and class B consists of output to be punched. Other classes are defined by the installation.

To route a data set through the output stream, and to assign it to class A, you would simply code SYSOUT=A in the DD statement. No other

parameters are necessary. A description of other parameters that can be coded appears in the publication IBM System/360 Operating System: Job Control Language.

In a system with the primary control program, you write a SYSOUT data set directly onto the system output device. This device is assigned to the output class by the operator; it may be either a unit record device or a magnetic tape unit. If it is a tape unit, the operator is responsible for transcribing the tape on a punch or printer. Depending on the output class and on the installation, the tape may be transcribed during a later shift or on a smaller, offline computing system.

In a system with MFT or MVT, you write a SYSOUT data set in one of two ways, as determined by the operator. Either you write the data set directly onto the system output device, or you write the data set into intermediate storage on a direct access device. In the latter case, a system output writer automatically copies your data set onto the system output device after your job has been completed. The system output device can be either a unit record device or a magnetic tape unit, as is true in systems with PCP.

Note: The following discussion assumes that, for systems with MFT and MVT, system output data sets are written into intermediate storage and copied by a SYSOUT writer. When these data sets are written directly onto the system output device, they are handled as described for systems with PCP.

OPENING A SYSOUT DATA SET

You open and close a SYSOUT data set in the same way as any other data set. If specified in an exit list, the data control block exit routine is entered in the usual manner. An exit list should not specify user label exits, because you cannot write labels on a unit record device.

If you observe certain restrictions, which are indicated below, you can create several SYSOUT data sets during a single job step.

Data Sets That Are Open Concurrently: In a system with the primary control program, only one of a group of concurrently open data sets can be assigned to an output class for which the SYSOUT device is a magnetic tape unit. If necessary, you can assign data sets to class A and a maximum of seven other output classes.

If a punch or printer is used as a SYSOUT device, any number of data sets can be assigned to one output class. The data control blocks for all these data sets can refer to the same DD statement. When printed or punched, the data sets appear as a single data set because their records form a single chronological sequence.

In a system with MFT or MVT, SYSOUT data sets must always be defined by separate DD statements. They can be assigned to the same output class or to different output classes. There is no special restriction on the number of output classes that can be used.

Data Sets That Are Not Open Concurrently: To avoid having two data sets open concurrently, you can open and close each data set as often as required. In a system with the primary control program, records of the two data sets will form a single chronological sequence if the data sets belong to the same output class.

In a system with MFT or MVT, records of two data sets do not form a chronological sequence, because you write each data set into a

separate area of intermediate direct access storage. When copied by a SYSOUT writer, data sets are written on the SYSOUT device in the order of their DD statements.

WRITING A SYSOUT DATA SET

To create a SYSOUT data set, you can use either the basic sequential or the queued sequential access method. You can write records in any format defined for the type of unit record device on which the data set is to be written. Record length must not exceed the maximum allowable for the device.

Under MFT or MVT, when you use the queued sequential access method (QSAM) with fixed blocked records or the basic sequential access method (BSAM), the DCB block size parameter does not have to be a multiple of logical record length (LRECL) if the block size is specified through the SYSOUT DD statement. Under these conditions, if block size is greater than LRECL but not a multiple of LRECL, block size is reduced to the nearest lower multiple of LRECL when the data set is opened. This feature allows a cataloged procedure to specify blocking for SYSOUT data sets, even though the user's LRECL is not known to the system until execution time. Therefore, the SYSOUT DD statement of the Go step of a compile-load-go procedure can specify block size without block size being a multiple of LRECL. For further information, refer to "Creating Data Sets in the Output Stream" in the publication IBM System/360 Operating System: Job Control Language User's Guide.

Because a SYSOUT data set may be written on a magnetic tape or direct access device, it must be device-independent. You should therefore omit the device dependency operand in the DCB macro instruction, or should code it as DEVD=DA.

Your SYNAD routine is entered on errors that occur when the data set is first written. In a system with the primary control program, it is entered when you write the data set on the system output device. In a system with MFT or MVT, it is entered when you write the data set into intermediate storage on a direct access device.

Your program is responsible for printing format, pagination, and header control. Use of control characters must be indicated in the usual way in the data control block. If you do not use control characters, a standard control is supplied in systems with MFT or MVT. When channel 12 is sensed, a printer will space one line and skip to channel 1; a card punch will select punch pocket 1.

In a system with MFT or MVT, cards can be punched only in EBCDIC mode.

CONCATENATING SEQUENTIAL AND PARTITIONED DATA SETS

Two or more sequential or partititioned data sets can be automatically retrieved by the system and processed successively as a single data set. This reading technique is known as concatenation. A maximum of 255 data sets (16, if partitioned) can be concatenated, but they must be used only for input. To save time when processing two consecutive data sets on a single volume, you specify LEAVE in your OPEN macro instruction. Concatenated data sets cannot be read backwards.

When data sets are concatenated, the system treats the group as a single data set and only one data extent block (DEB) is constructed. Thus, it is important to consider the characteristics of the

individual data sets which are being concatenated. Data sets with like characteristics are those which may be processed correctly using the same data control block (DCB), input/output block (IOB), and channel program. Any exception makes them "unlike". The system must be informed if "unlike" data sets are concatenated. This is accomplished by modifying the DCBOFLGS field of the data control block. The indication must be made before the end of the current data set is reached. You must set bit 4 to one by using the instruction OI DCBOFLGS,X'08' as described in "Modifying the Data Control Block". If the DCBOFLGS field is X'08', end-of-volume processing for each data set will issue a CLOSE for the data set just read and an OPEN for the next concatenated data set. This procedure causes the updating of the fields in the DCB, and if necessary, the building of a new IOB and channel program. Unless you have some way of determining the characteristics of the next data set before it is opened, you should not reset the field to indicate "like" characteristics during processing.

When "unlike" data sets have been concatenated, we urge that you do not issue multiple input requests, i.e., a series of READ or GET macro instructions in your program. Otherwise, you will have to arrange some way to determine which requests have been completed and which must be reissued. In any case, the GET or READ macro instruction that detected the end of data set will have to be reissued. Figure 37 illustrates a possible routine for determining when a GET or READ must be reissued. This restriction does not apply to "like" data sets since no open or close operation is necessary between data sets.

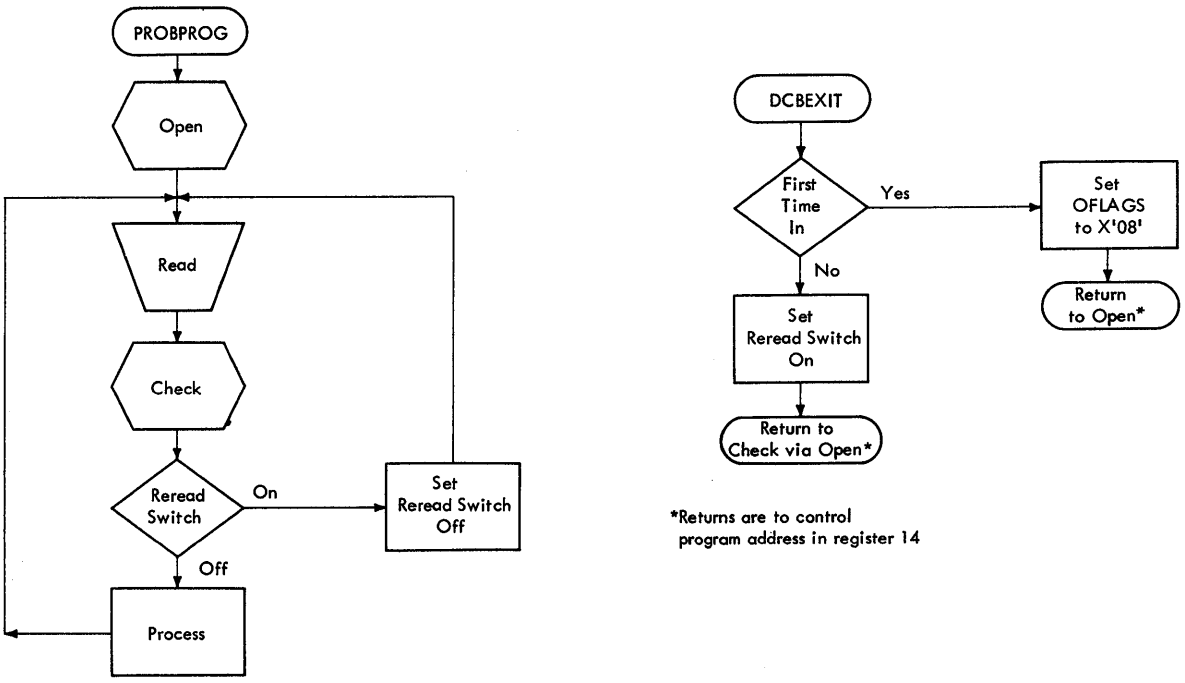


Figure 37. Reissuing a READ for "Unlike" Concatenated Data Sets

When the change is made from one data set to another, label exits are taken as required; automatic volume switching is also performed for multiple volume data sets. Your end-of-data-set (EODAD) routine is not entered until the last data set has been processed. An exception to this arises with partitioned data sets. Your EODAD routine receives control at the end of each member. At that time, you can process the next member or close the data set.

Further discussion and examples of concatenated data sets are contained in the publication IBM System/360 Operating System: Job Control Language Reference.

CATALOGING DATA SETS

To provide the cataloging facilities of the operating system, a catalog is created that is itself a data set residing on one or more direct access volumes. It is organized into levels of indexes that connect the data set names to corresponding volumes and data set sequence numbers. For each level of qualification in the data set name, there is an index group in the catalog.

The highest level of the catalog resides on the system residence volume. The volume table of contents (VTOC) contains an entry for the data set control block (DSCB) defining the catalog and its highest level index, the volume index. The lowest level index contains the simple name of the data set and the number of the volume on which it resides.

The complete catalog can exist on the system residence volume, or you can specify that parts of it be constructed on other volumes. Any volume containing part of the catalog is called a control volume. The use of control volumes allows data sets that are functionally related to be cataloged separately. There are several advantages:

- Control volumes can be moved from one processing system to another.
- System residence requirements can be reduced by placing seldom used indexes on a control volume.

For any given data set, only one level of control volume, other than the system residence volume, can be used. Notice that in Figure 38, INDEX E, which is the highest level on the control volume, has an entry in both volume indexes.

The same type of cataloging facilities are available for maintaining generation data groups. Cataloging each new generation with a unique name would be both inconvenient and inefficient. By cataloging individual data sets in a chronological collection by number, the entire collection can be stored under a single data set name.

Each update of the data set is called a generation; the number associated with it is called a generation number. A generation data group is the entire collection of chronologically related data sets that can be referred to by the same data set name. A particular generation can be referred to by either the absolute generation name or relative generation number of the data set.

ABSOLUTE GENERATION NAME: The operating system assigns each data set in the generation data group an absolute generation name in the form GggggVvv:

- gggg is an unsigned, four digit, decimal generation number.
- vv is an unsigned, two digit, decimal version number.

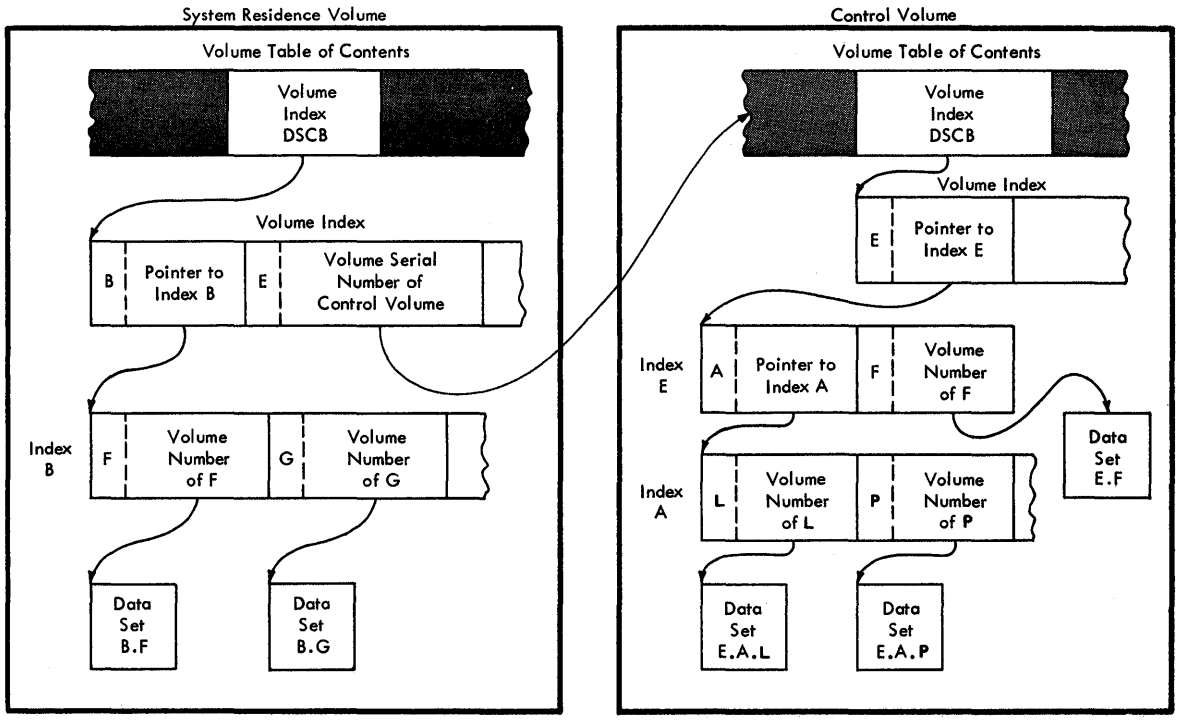


Figure 38. Catalog Structure on Two Volumes

The generation number indicates how far removed the data set is from the original generation. The version number indicates how many times the associated generation has been replaced. Only the most recent version of a specific generation is retained.

Generation Increment: You can specify the increment by which the generation number is changed. For example, if you request a current generation G0013V04 and an increment of 2, the new generation would be assigned the absolute generation name G0015V00.

Version Increment: When you replace the same generation with a new version, it is your responsibility to assign the new, nonzero version number.

CONCATENATED GENERATIONS: You can request a concatenation of all existing data sets in the generation data group, starting with the most recent and ending with the oldest, by specifying only the data set name.

RELATIVE GENERATION NUMBER: Rather than request a data set by its absolute generation number, you can refer to it relative to the most recent generation, i.e., DSNAME=dsname(0). Those immediately preceding the most recent are then identified as -1, -2, etc. New generations are created by referring to them as DSNAME=name(+1),(+2),(+3), etc. The last of these is cataloged as (0) and the other generations in the catalog are adjusted accordingly at the end of the job.

ENTERING A DATA SET NAME IN THE CATALOG

The catalog structure, including all levels of indexes, is initially created or modified by the system utility program IEHPROGM. A data set name can then be entered if the proper index levels of the name exist.

For example, if a data set named A.B.C is to be cataloged, the volume index on the system residence volume must have an index entry for index A, which must point to an index B. When the data set A.B.C is cataloged, C is entered into index B along with the volume serial number where data set A.B.C resides. The cataloging request is entered as:

```
//ddname DD DSNAME=A.B.C,DISP=(,CATLG)
```

ENTERING A GENERATION DATA GROUP IN THE CATALOG

A data set that is part of a generation data group is represented in the catalog by an additional level of indexing that contains an entry for each generation. The system utility program IEHPROGM is used to create the index levels and to instruct the system as to how the generations are to be maintained.

CONTROL OF CONFIDENTIAL DATA -- PASSWORD PROTECTION

In addition to the usual label protection that prevents opening a data set without the correct data set name, the operating system provides a data set security facility that prevents unauthorized access to confidential data. A security protected data set cannot be made available for processing until a password is entered by the operator. If an incorrect password is entered twice, the job is terminated by the system.

You can request password protection when the data set is created. The system sets the data set security byte in the Standard Data Set Label 1 as shown in the publication, IBM System/360 Operating System Tape Labels. Once security protection has been requested, it cannot be removed without recreating the data set and scratching the protected data set.

Each protected data set has at least one entry in a catalog named PASSWORD that must be created on the system residence volume. Each entry in the password data set consists of a 44-byte data set name field and an 8-byte password field. The next 80-byte record contains a 2-byte binary counter that is incremented by one each time the protected data set is opened successfully. The third byte is used to indicate that the processing program can read, write, or both read and write records on the protected data set. The remaining 77 bytes can be used at the discretion of your installation.

The password data set can also be protected by a master password contained in one of its entries. A complete description of password protection is contained in the publication IBM System/360 Operating System: System Programmer's Guide.

Appendix A: Direct Access Labels

Only standard label formats are used on direct access volumes. Volume, data set, and optional user labels are used (see Figure 39). In the case of direct access volumes, the data set label group is the data set control block (DSCB).

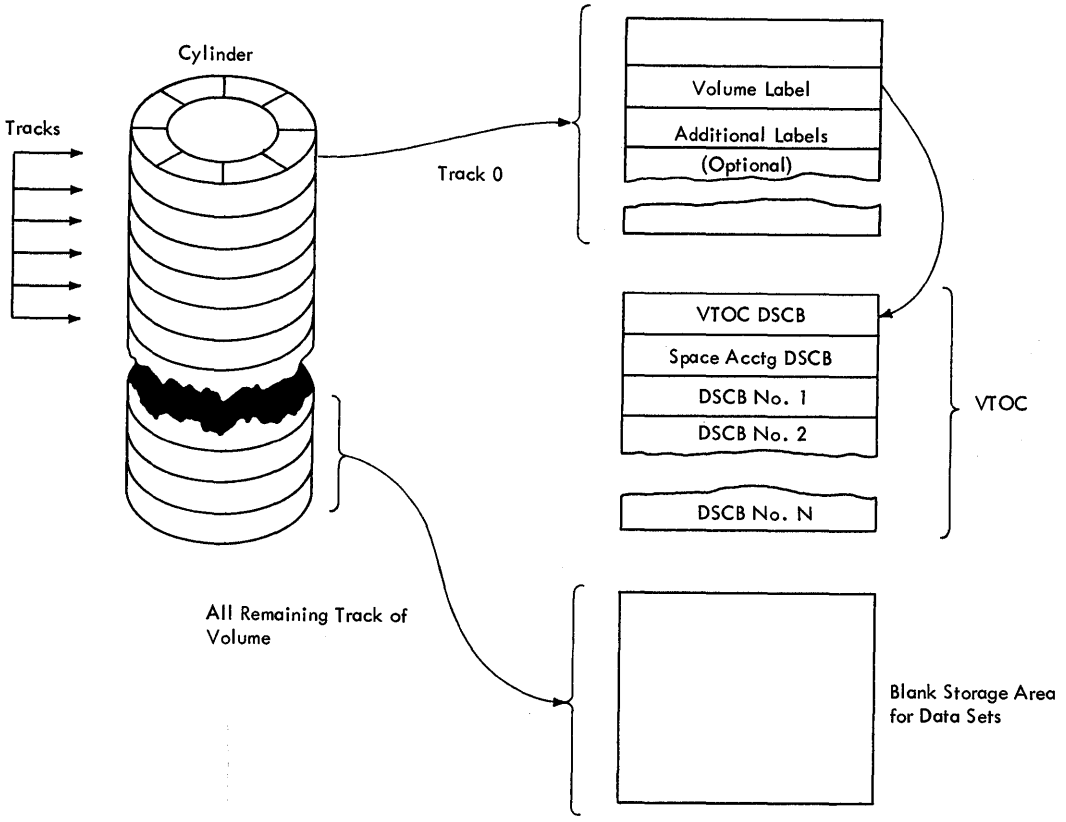


Figure 39. Direct Access Labeling

VOLUME LABEL GROUP

The volume label group immediately follows the initial program loading (IPL) records on track 0 (of cylinder 0) of the volume. It consists of the initial volume label plus a maximum of seven additional volume labels. The initial volume label identifies a volume and its owner, and is used to verify that the correct volume is mounted. It can also be used to prevent use of the volume by unauthorized programs. The additional labels are processed by means of an installation routine that is incorporated into the system.

The format of the direct access volume label group is shown in Figure 40.

DIRECT ACCESS VOLUME LABEL FORMAT

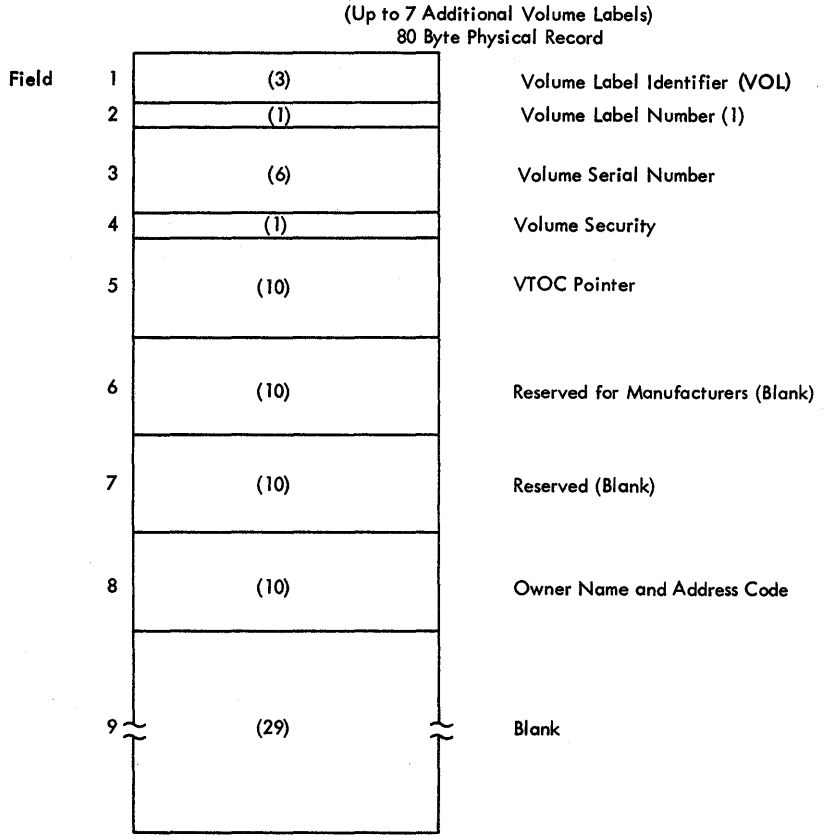


Figure 40. Initial Volume Label

Volume Label Identifier (VOL): Field 1 contains the initial volume label.

Volume Label Number (1): Field 2 identifies the relative position of the volume label in a volume label group. It must be written as 1.

The operating system identifies an initial volume label when, in reading the initial record, it finds that the first four characters of the record are VOL1.

Volume Serial Number: Field 3 contains a unique identification code assigned when the volume enters the system. You can place the code on the external surface of the volume for visual identification. The code is normally numeric (000001-999999), but may be any six alphameric characters.

Volume Security: Field 4 is reserved for future use by installation that wish to provide security at the volume level. It must be written as 0.

VTOC Pointer: Field 5 of direct access volume label 1 contains the address of the volume table of contents (VTOC).

Reserved for Manufacturers: Field 6 is reserved for future standardization purposes. Leave it blank.

Reserved: Field 7 is reserved for future developmental purposes. Leave it blank.

Owner Name and Address Code: Field 8 contains a unique identification of the owner of the volume.

All of the bytes in Field 9 are left blank.

DATA SET CONTROL BLOCK (DSCB) GROUP

The system automatically constructs a DSCB when space is requested for a data set on a direct access volume. Each data set on a direct access volume has a corresponding data set control block to describe its characteristics. The DSCB appears in the volume table of contents (VTOC) and contains operating system data, device-dependent information, and data set characteristics, in addition to space allocation and other control information. The format of the DSCB is illustrated in IBM System/360 Operating System: System Control Blocks.

USER LABEL GROUPS

User header and trailer label groups can be included with data sets of physically sequential or direct organization. The labels in each group have the format shown in Figure 33.

Each group can include up to eight labels, but the space required for both groups must not be more than one track on a direct access device. The current minimum track size allows a maximum of eight labels, including both header and trailer labels. Consequently, a program becomes device-dependent (among direct access devices) when it creates more than eight labels.

If user labels are specified in the DD statement (LABEL=SUL), an additional track is normally allocated when the data set is created. No additional track is allocated when specific tracks are requested (SPACE=(ABSTR,...)), or when tracks allocated to another data set are requested (SUBALLOC=...). In either case, labels are written on the first track that is allocated.

User Header Label Group: The operating system writes these labels as directed by the problem program recording the data set. The first four characters of the user header label must be UHL1,..., UHL8; you can specify the remaining 76 characters. When the data set is read, the operating system makes the user header labels available to the problem program for processing.

User Trailer Label Group: These labels are recorded (and processed) as explained in the preceding text for user header labels, except that the first four characters must be UTL1,...., UTL8.

USER HEADER AND TRAILER LABEL FORMAT

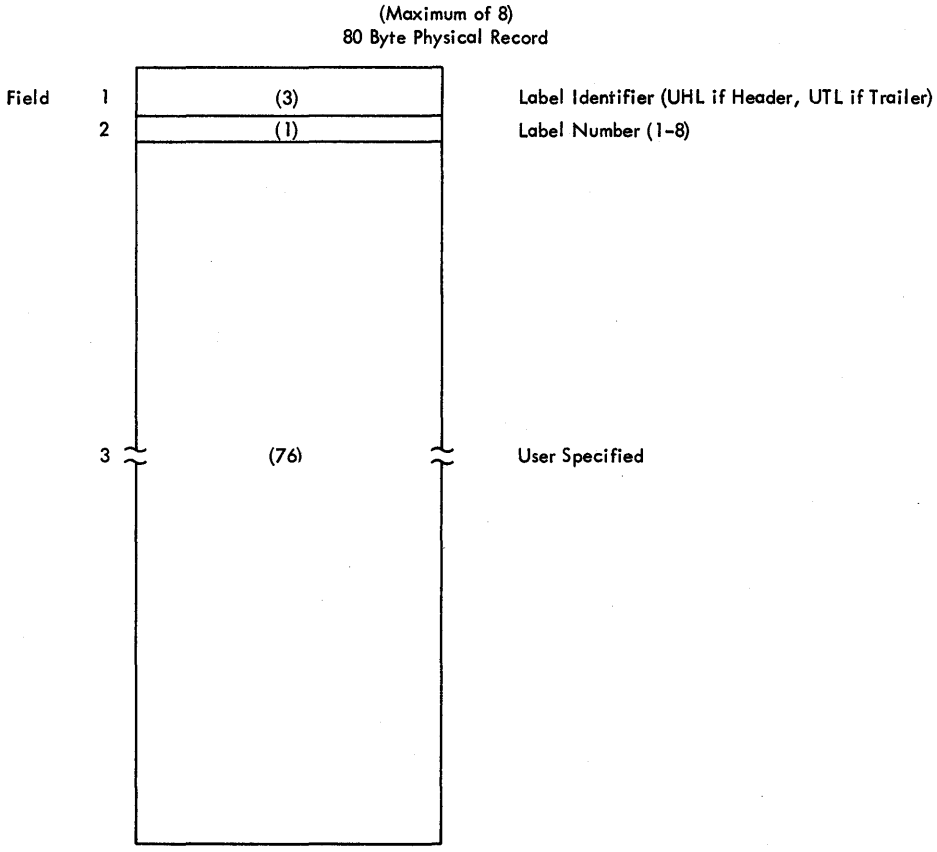


Figure 41. User Header and Trailer Labels

Label Identifier: Field 1 indicates that this is a user header label (UHL). UTL indicates a user trailer label.

Label Number: Field 2 identifies the relative position (1-8) of the label within the user label group.

User Specified: Field 3 (76 bytes).

Appendix B: Control Characters

As an optional feature, all record formats may include a control character in each logical record. This control character will be recognized and processed if a data set is being written to a printer or punch.

For format-F and -U records this character is the first byte of the logical record.

For format-V records it must be the fifth byte of the logical record, immediately following the record descriptor word.

Two options are available. If either option is specified in the data control block, the character must appear in every record and other line spacing or stacker selection options also specified in the data control block are ignored.

Machine Code

You can specify in the data control block that the machine code control character has been placed in each logical record. If the record is to be written, the appropriate byte must contain the command code bit configuration specifying both the write and the desired carriage or stacker select operation. If the record is not to be written, the byte can specify any command other than write.

Command codes for specific devices are contained in IBM System Reference Library publications describing the control units or devices.

Extended American National Standard Code for Information Interchange

In place of machine code, you can specify control characters defined by the American National Standards Institute, Inc. (ANSI). These characters must be represented in EBCDIC.

The extended American National Standard Code for Information Interchange (ASCII) is as follows:

Code	Action Before Writing Record
b	Space one line before printing (blank code)
0	Space two lines before printing
-	Space three lines before printing
+	Suppress space before printing
1	Skip to channel 1
2	Skip to channel 2
3	Skip to channel 3
4	Skip to channel 4
5	Skip to channel 5
6	Skip to channel 6
7	Skip to channel 7
8	Skip to channel 8
9	Skip to channel 9
A	Skip to channel 10
B	Skip to channel 11
C	Skip to channel 12
V	Select punch pocket 1
W	Select punch pocket 2

These control characters include those defined by ANSI FORTRAN. If any other character is specified, it is interpreted as 'b' or V, depending on the device being used; no error indication is returned.

Index

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 and Data Management Services

GC28-6646-3

- Your comments, accompanied by answers to the following questions, help us produce better publications for your use. If your answer to a question is "No" or requires qualification, please explain in the space provided below. Comments and suggestions become the property of IBM.

	Yes	No
• Does this publication meet your needs?	☐	☐
• Did you find the material:		
Easy to read and understand?	☐	☐
Organized for convenient use?	☐	☐
Complete?	☐	☐
Well illustrated?	☐	☐
Written for your technical level?	☐	☐

- What is your occupation? _____
- How do you use this publication?

As an introduction to the subject?	☐	As an instructor in a class?	☐
For advanced knowledge of the subject?	☐	As a student in a class?	☐
For information about operating procedures?	☐	As a reference manual?	☐

Other _____

- Please give specific page and line references with your comments when appropriate.

COMMENTS

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

YOUR COMMENTS, PLEASE . . .

This publication is one of a series which serves as a reference source 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. 2078
SAN JOSE, CALIF.

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

POSTAGE WILL BE PAID BY . . .

IBM Corporation
Monterey & Cottle Rds.
San Jose, California
95114

Attention: Programming Publications, Dept. D78

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)