TOPS-20
Monitor Calls User's Guide

Electronically Distributed

This manual describes the use of TOPS-20 monitor calls, which provide user programs with system services such as input/output, process control, file handling, and device control.

This manual supersedes the TOPS-20 Monitor Calls User's Guide published in June 1988. The order number for that document, AA-D859DM-TM, is obsolete.

Change bars in the margins indicate material that has been added or changed since the previous printing of this manual.

Operating System: TOPS-20 Version 7.0

digital equipment corporation maynard, massachusetts

TOPS-20 Software Update Tape No. 04, November 1990

First Printing, May 1976
Revised, April 1982
Revised, September 1985
Revised, June 1988
Revised, November 1990

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may only be used or copied in accordance with the terms of such license.

No responsibility is assumed for the use or reliability of software on equipment that is not supplied by Digital Equipment Corporation or its affiliated companies.

Copyright C 1976, 1982, 1985, 1988, 1990 Digital Equipment Corporation.

All Rights Reserved.

The following are trademarks of Digital Equipment Corporation:

CI	DECtape	LA50	SITGO-10
DDCMP	DECUS	LN01	TOPS-10
DEC	DECwriter	LN03	TOPS-20
DECmail	DELNI	MASSBUS	TOPS-20AN
DECnet	DELUA	PDP	UNIBUS
DECnet-VAX	HSC	PDP-11/24	UETP
DECserver	HSC-50	PrintServer	VAX
DECserver 100	KA10	PrintServer 40	VAX/VMS
DECserver 200	KI	Q-bus	VT50
DECsystem-10	KL10	ReGIS	
DECSYSTEM-20	KS10	RSX	d i g i t a l

CONTENTS

PREFACE

CHAPTER 1 INTRODUCTION

1.1 OVERVIEW 1-1
1.2 MONITOR CALLS 1-2
1.2.1 Calling Sequence 1-3
1.2.2 Error Returns 1-4
1.3 PROGRAM ENVIRONMENT 1-6

CHAPTER 2 INPUT AND OUTPUT USING THE TERMINAL

2.1 OVERVIEW 2-1
2.2 PRIMARY I/O DESIGNATORS 2-2
2.3 PRINTING A STRING 2-3
2.4 READING A NUMBER 2-4
2.5 WRITING A NUMBER 2-5
2.6 INITIALIZING AND TERMINATING THE PROGRAM 2-7
2.6.1 RESET% Monitor Call 2-8
2.6.2 HALTF% Monitor Call 2-8
2.7 READING A BYTE 2-8
2.8 WRITING A BYTE 2-8
2.9 READING A STRING 2-9
2.10 SUMMARY 2-14

CHAPTER 3 USING FILES

3.1 OVERVIEW 3-1
3.2 JOB FILE NUMBER 3-2
3.3 ASSOCIATING A FILE WITH A JFN 3-3
3.3.1 GTJFN% Monitor Call 3-4
3.3.1.1 Short Form of GTJFN% 3-4
3.3.1.2 Long Form of GTJFN% 3-12
3.3.1.3 Summary of GTJFN% 3-15
3.4 OPENING A FILE 3-16
3.4.1 OPENF% Monitor Call 3-16
3.5 TRANSFERRING DATA 3-19
3.5.1 File Pointer 3-20
3.5.2 Source and Destination Designators 3-20
3.5.3 Transferring Sequential Bytes 3-21
3.5.4 Transferring Strings 3-22
3.5.5 Transferring Nonsequential Bytes 3-24
3.5.6 Mapping Pages 3-24
3.5.6.1 Mapping File Pages to a Process 3-26
3.5.6.2 Mapping Process Pages to a File 3-27
3.5.6.3 Unmapping Pages in a Process 3-28
3.5.7 Mapping File Sections to a Process 3-28
3.6 CLOSING A FILE 3-30
3.6.1 CLOSF% Monitor Call 3-30
3.7 ADDITIONAL FILE I/O MONITOR CALLS 3-31
3.7.1 GTSTS% Monitor Call 3-31
3.7.2 JFNS% Monitor Call 3-33
3.7.3 GNJFN% Monitor Call 3-36
3.8 SUMMARY 3-40
3.9 FILE EXAMPLES 3-40

CHAPTER 4 USING THE SOFTWARE INTERRUPT SYSTEM

4.1 OVERVIEW 4-1
4.2 INTERRUPT CONDITIONS 4-4
4.3 SOFTWARE INTERRUPT CHANNELS AND PRIORITIES 4-4
4.4 SOFTWARE INTERRUPT TABLES 4-6
4.4.1 Specifying the Software Interrupt Tables 4-6
4.4.2 Channel Table 4-7
4.4.3 Priority Level Table 4-8
4.5 ENABLING THE SOFTWARE INTERRUPT SYSTEM 4-9
4.6 ACTIVATING INTERRUPT CHANNELS 4-9
4.7 GENERATING AN INTERRUPT 4-10
4.8 PROCESSING AN INTERRUPT 4-10
4.8.1 Dismissing an Interrupt 4-11
4.9 TERMINAL INTERRUPTS 4-12
4.10 ADDITIONAL SOFTWARE INTERRUPT MONITOR CALLS . . . 4-14
4.10.1 Testing for Enablement 4-14
4.10.2 Obtaining Interrupt Table Addresses 4-15
4.10.2.1 The RIR% Monitor Call 4-15
4.10.2.2 The XRIR% Monitor Call 4-15
4.10.3 Disabling the Interrupt System 4-16
4.10.4 Deactivating a Channel 4-16
4.10.5 Deassigning Terminal Codes 4-17
4.10.6 Clearing the Interrupt System 4-17
4.11 SUMMARY 4-17
4.12 SOFTWARE INTERRUPT EXAMPLE 4-18

CHAPTER 5 PROCESS STRUCTURE

5.1 USES FOR MULTIPLE PROCESSES 5-2
5.2 PROCESS COMMUNICATION 5-3
5.2.1 Direct Process Control 5-4
5.2.2 Software Interrupts 5-4
5.2.3 IPCF and ENQ/DEQ Facilities 5-4
5.2.4 Memory Sharing 5-5
5.3 PROCESS IDENTIFIERS 5-5
5.4 OVERVIEW OF MONITOR CALLS FOR PROCESSES 5-7
5.5 CREATING A PROCESS 5-8
5.5.1 Process Capabilities 5-11

5.6 SPECIFYING THE CONTENTS OF THE ADDRESS SPACE OF A PROCESS 5-11
5.6.1 GET% Monitor Call 5-11
5.6.2 PMAP% Monitor Call 5-14
5.7 STARTING AN INFERIOR PROCESS 5-15
5.8 INFERIOR PROCESS TERMINATION 5-16
5.9 INFERIOR PROCESS STATUS 5-17
5.10 PROCESS COMMUNICATION 5-19
5.11 DELETING AN INFERIOR PROCESS 5-20
5.12 PROCESS EXAMPLES 5-21

CHAPTER 6 ENQUEUE/DEQUEUE FACILITY

6.1 OVERVIEW 6-1
6.2 RESOURCE OWNERSHIP 6-2
6.3 PREPARING FOR THE ENQ/DEQ FACILITY 6-3
6.4 USING THE ENQ/DEQ FACILITY 6-6
6.4.1 Requesting Use of a Resource 6-6
6.4.1.1 ENQ% Functions 6-6
6.4.1.2 ENQ% Argument Block 6-8
6.4.2 Releasing a Resource 6-12
6.4.2.1 DEQ% Functions 6-13
6.4.2.2 DEQ% Argument Block 6-14
6.4.3 Obtaining Information About Resources 6-14
6.5 SHARER GROUPS 6-17
6.6 AVOIDING DEADLY EMBRACES 6-19

CHAPTER 7 INTER-PROCESS COMMUNICATION FACILITY

7.1 OVERVIEW 7-1
7.2 QUOTAS 7-1
7.3 PACKETS 7-2
7.3.1 Flags 7-3
7.3.2 PIDs 7-5
7.3.3 Length and Address of Packet Data Block 7-6
7.3.4 Directories and Capabilities 7-6
7.3.5 Packet Data Block 7-6
7.4 SENDING AND RECEIVING MESSAGES 7-7
7.4.1 Sending a Packet 7-7
7.4.2 Receiving a Packet 7-9
7.5 SENDING MESSAGES TO <SYSTEM>INFO 7-12
7.5.1 Format of <SYSTEM>INFO Requests 7-13
7.5.2 Format of <SYSTEM>INFO Responses 7-14
7.6 PERFORMING IPCF UTILITY FUNCTIONS 7-15

CHAPTER 8 USING EXTENDED ADDRESSING

8.1 OVERVIEW 8-1
8.2 ADDRESSING MEMORY AND ACS 8-2
8.2.1 Instruction Format 8-3
8.2.2 Indexing 8-4
8.2.3 Indirection 8-5
8.2.3.1 Instruction Format Indirect Word (IFIW) . . . 8-5
8.2.3.2 Extended Format Indirect Word (EFIW) 8-6
8.2.3.3 Macros for Indirection 8-6
8.2.4 AC References 8-6
8.2.5 Extended Addressing Examples 8-7
8.2.6 Immediate Instructions 8-8
8.2.6.1 XMOVEI 8-8
8.2.6.2 XHLLI 8-9
8.2.7 Other Instructions 8-9
8.2.7.1 Instructions that Affect the PC 8-10
8.2.7.2 Stack Instructions 8-10
8.2.7.3 Byte Instructions 8-10
8.3 USING MONITOR CALLS 8-11
8.3.1 Mapping Memory 8-12
8.3.1.1 Mapping File Sections to a Process 8-12
8.3.1.2 Mapping Process Sections to a Process . . . 8-13
8.3.1.3 Creating Sections 8-14
8.3.1.4 Unmapping a Process Section 8-15
8.3.2 Starting a Process in Any Section 8-16
8.3.3 Setting the Entry Vector in Any Section . . . 8-16
8.3.4 Obtaining Information About a Process 8-17
8.3.4.1 Memory Access Information 8-17
8.3.4.2 Entry Vector Information 8-19
8.3.4.3 Page-Failure Information 8-19
8.3.5 Program Data Vectors 8-19
8.3.5.1 Manipulating PDV Addresses 8-20
8.3.5.2 PDV Names 8-20
8.3.5.3 Version Number 8-21
8.4 MODIFYING EXISTING PROGRAMS 8-21
8.4.1 Data Structures 8-21
8.4.1.1 Index Words 8-22
8.4.1.2 Indirect Words 8-22
8.4.1.3 Stack Pointers 8-22
8.5 WRITING MULTISECTION PROGRAMS 8-22

INDEX

FIGURES

4-1 Basic Operational Sequence of the Software Interrupt System 4-3
6-1 Deadly Embrace Situation 6-5
6-2 Use of Sharer Groups 6-18
7-1 IPCF Packet 7-2
8-1 Program Counter Address Fields 8-2
8-2 Instruction Word Address Fields 8-4
8-3 Instruction Format Indirect Word 8-5

8-4 Extended Format Indirect Word 8-6

TABLES

2-1 NOUT% Format Option 2-6
2-2 RDTTY% Control Bits 2-10
3-1 Standard System Values for File Specifications . . 3-3
3-2 GTJFN% Flag Bits 3-5
3-3 Bits Returned on GTJFN% Call 3-10
3-4 Long Form GTJFN% Argument Block 3-13
3-5 OPENF% Access Bits 3-17
3-6 PMAP% Access Bits 3-26
3-7 SMAP% Access Bits 3-29
3-8 CLOSF% Flag Bits 3-30
3-9 Bits Returned on GTSTS% Call 3-31
3-10 JFNS% Format Options 3-34
3-11 GNJFN% Return Bits 3-37
4-1 Software Interrupt Channel Assignments 4-5
4-2 Terminal Codes and Conditions 4-12
5-1 Process Handles 5-6
5-2 Inferior Process Characteristic Bits 5-9
5-3 GET% Flag Bits 5-12
5-4 GET% Argument Block 5-13
5-5 GET% Argument Block Flags 5-13
5-6 Process Status Word 5-17
5-7 RFSTS% Status-Return Block 5-18
6-1 ENQ% Functions 6-7
6-2 ENQ% Argument Block 6-8
6-3 Lock Specification Flags 6-10
6-4 DEQ% Functions 6-13
6-5 DEQ% Argument Block 6-14
6-6 ENQC% Flag Bits 6-16
7-1 Packet Descriptor Block Flags 7-3
7-2 Flags Meaningful on a MSEND% Call 7-8
7-3 Flags Meaningful on a MRECV% Call 7-10
7-4 MRECV% Return Bits 7-12
7-5 <SYSTEM>INFO Functions and Arguments 7-14
7-6 <SYSTEM>INFO Responses 7-15
7-7 MUTIL% Functions 7-16

PREFACE

The TOPS-20 Monitor Calls User's Guide is written for the assembly language user who is unfamiliar with the DECSYSTEM-20 monitor calls. The manual introduces the user to the functions that he can request of the monitor from within his assembly language programs. The manual also teaches him how to use the basic monitor calls for performing these functions.

This manual is not a reference document, nor is it complete documentation of the entire set of monitor calls. It is organized according to functions, starting with the simple and proceeding to the more advanced.

Each chapter should be read from beginning to end. A user who skips around in his reading will not gain the full benefit of this manual. Once the user has a working knowledge of the monitor calls in this document, he should then refer to the TOPS-20 Monitor Calls Reference Manual for the complete descriptions of all the calls.

To understand the examples in this manual, the user must be familiar with the MACRO language and the DECSYSTEM-20 machine instructions. The TOPS-20 MACRO Assembler Reference Manual documents the MACRO language. The TOPS-20 LINK Reference Manual describes the linking loader. The DECsystem-10/DECSYSTEM-20 Processor Reference Manual contains the information on the machine instructions. These three manuals should be used together with the Monitor Calls User's Guide, and should be referred to when questions arise on the MACRO language or the instruction set. Another useful reference is Introduction to DECSYSTEM-20 Assembly Language Programming by Ralph E. Gorin, published by the Digital Press. It provides a thorough treatment of assembly language programming for the DECSYSTEM-20, emphasizing the analysis of programs and various methods of program synthesis.

In addition, some of the examples in this manual contain macros and symbols (MOVX, TMSG, JSERR, or JSHLT, for example) from the MACSYM system file. This file is a universal file of definitions available

to the user as a means of producing consistent and readable programs.

Finally, the user should be familiar with the TOPS-20 Command Language to enter and run the examples. The TOPS-20 User's Guide describes the TOPS-20 commands and system programs. The TOPS-20 Commands Reference Manual describes all operating system commands available to the nonprivileged user of TOPS-20.

CHAPTER 1

INTRODUCTION

1.1 OVERVIEW

A program written in MACRO assembly language consists of a series of statements, each statement usually corresponding to one or more machine language instructions. Each statement in the MACRO program may be one of the following types:

1. A MACRO assembler directive, or pseudo-operation (pseudo-op), such as SEARCH or END. These pseudo-ops are commands to the MACRO assembler and are performed when the program is assembled. Refer to the _DECSYSTEM-20 MACRO Assembler Reference Manual_ for detailed descriptions of the MACRO pseudo-ops.

2. A MACRO assembler direct assignment statement. These statements are in the form

   symbol=value

   and are used to assign a specific value to a symbol. Assignment statements are processed by the MACRO assembler when the program is assembled. These statements do not generate instructions or data in the assembled program.

3. A MACRO assembler constant declaration statement, such as

   ONE: EXP 1

   These statements are processed when the program is assembled.

4. An instruction mnemonic, or symbolic instruction code, such as MOVE or ADD. These symbolic instruction codes represent the operations performed by the central processor when the program is executed. Refer to the _DECsystem-10/DECSYSTEM-20 Processor Reference Manual_ for detailed descriptions of the symbolic instruction codes.

5. A monitor call, or JSYS, such as RESET or BIN. These calls are commands to the monitor and are performed when the program is executed. This manual describes the commonly-used monitor calls. However, the user should refer to the _TOPS-20 Monitor Calls Reference Manual_ for detailed descriptions of all the calls.

When the MACRO program is assembled, the MACRO assembler processes the statements in the program by

- o translating symbolic instruction codes to binary codes.
- o relating symbols to numeric values.
- o assigning relocatable or absolute memory addresses.

The MACRO assembler also converts each symbolic call to the monitor into a Jump-to-System (JSYS) instruction.

1.2 MONITOR CALLS

Monitor calls are used to request monitor functions, such as input or output of data (I/O), error handling, and number conversions, during the execution of the program. These calls are accomplished with the JSYS instruction (operation code 104), where the address portion of the instruction indicates the particular function.

Each monitor call has a predefined symbol indicating the particular monitor function to be performed (for example, OPENF% to indicate opening a file). The symbols are defined in a system file called MONSYM. Monitor calls defined in Release 4 and later require a percent sign (%) as the final character in the call symbol. Monitor calls defined prior to Release 4 do not require the %, but do accept it. The current convention is that all monitor calls use the % as part of the call symbol. This manual follows that convention. To use the symbols and to cause them to be defined correctly, the user's program must contain the statement

    SEARCH MONSYM

at the beginning of the program. During the assembly of the program, the assembler replaces the monitor call symbol with an instruction containing the operation code 104 in the left half and the appropriate function code in the right half.

Arguments for a JSYS instruction are placed in accumulators (ACs). Any data resulting from the execution of the JSYS instruction are returned in the accumulators or in an address in memory to which an accumulator points. Therefore, before the JSYS instruction can be executed, the appropriate arguments must be placed in the specific accumulators.

The system file MACSYM.MAC contains a number of useful macros for the assembly language programmer. To use MACSYM macros, the user's program must contain the statements

```
     SEARCH MACSYM
     .REQUIRE SYS:MACREL      ;include support routines
```

at the beginning of the program. Since most bits defined for use with the monitor have symbolic names, macros enable the programmer to utilize these bits without knowledge of their exact position. Several MACSYM macros that are especially valuable to the TOPS-20 assembly language programmer are MOVX, TXnn (where nn indicates one of the 64 test instructions provided by the hardware), and FLD. MOVX loads an AC with a constant using the proper MOVE instructions, depending on the constant's position in the word. The TXnn macros expand to allow all combinations of modification and testing to be defined. For example

```
     TXNN AC1,GS%EOF
```

tests AC1 for the presence of GS%EOF, no modification, and skip if not equal to zero. This instruction will work regardless of the actual bit position of GS%EOF. The FLD macro causes a value to be right justified in a field. For example

```
     FLD(7,OF%BSZ)
```

places the value 7 in position OF%BSZ, right justified at bit 5 (OF%BSZ is defined as bits 0-5). These macros will be used consistently throughout this document.

1.2.1 Calling Sequence

Arguments for the calls are placed in accumulators 1 through 4 (AC1-AC4). If more than four arguments are required for a particular call, the arguments are placed in a list to which an accumulator points. The arguments for the calls are specific bit settings or values. These bit settings and values are defined in MONSYM with symbol names, which can be used in the program. In fact, it is recommended that the user write his program using symbols whenever possible. This makes the program easier to read by another user. Use of symbols also allows the values of the symbols to be redefined without requiring the program to be changed. In this manual, the arguments for the monitor calls are described with both the bit settings and the symbol names. All program examples are written using the symbol names.

The set of instructions that place the arguments in the accumulators is followed by one line of code giving the particular monitor call symbol. During the program's execution, control is transferred to the monitor when this line of code is reached.

1.2.2 Error Returns

TOPS-20 provides a consistent way to handle all JSYS errors. For most monitor calls upon a successful return, the instruction following the call is executed. If an error occurs during the execution of the call, the monitor examines the instruction following the call. If the instruction is a JUMP instruction with the AC field specified as 12-17, the monitor transfers control to a user-specified address. If the instruction is not a JUMP instruction, the monitor generates an illegal instruction trap indicating an illegal instruction, which the user's program can process via the software interrupt system (refer to Chapter 4). If the user's program is not prepared to process the instruction trap, the program execution halts, and a message is output stating the reason for failure.

To place a JUMP instruction in his program, the user can include a statement using one of six predefined symbols. These symbols are

```
     ERJMPR  address   (= JUMP 12,address)
     ERCALR  address   (= JUMP 13,address)
     ERJMPS  address   (= JUMP 14,address)
     ERCALS  address   (= JUMP 15,address)
     ERJMP   address   (= JUMP 16,address)
     ERCAL   address   (= JUMP 17,address)
```

and cause the assembler to generate a JUMP instruction. The JUMP instruction is a non-operation instruction (that is, a no-op) as far as the hardware is concerned. However, the monitor executes the JUMP instruction by transferring control to the address specified, which is normally the beginning of an error processing routine written by the user. If the user includes the ERJMP symbol, control is transferred as though a JUMPA instruction had been executed, and control does not return to his program after the error routine is finished. If the user includes the ERCAL symbol, control is transferred as though a PUSHJ 17, address instruction had been executed. If the error routine executes a POPJ 17, instruction, control returns to the user's program at the location following the ERCAL.

If the user includes the ERJMPR symbol, the monitor behaves the same as it would if the ERJMP symbol had been included, except that the last error encountered by the process is stored in the user's AC1. (Refer to Appendix B of the TOPS-20 Monitor Calls Reference Manual for the list of error codes, mnemonics, and message strings.) The ERCALR symbol functions the same as ERCAL except the error code encountered is returned in the user's AC1. ERJMPS and ERCALS function similarly except the monitor suppresses the storing of the error code in the

user's AC1. Instead, AC1 is preserved and contains either the original contents from when the monitor call was given, or a partially updated value prior to the error.

Prior to the implementation of the ERJMP/ERCAL facilities, certain monitor calls returned control to the user's program at various locations after the calling address. Approximately one third of the JSYSs return to the +1 address only on failure, and to the location immediately following that (the +2 address) on successful execution of the call. A few calls return +1, +2, or +3, dependent on varying conditions of success or failure (for examples, see ERSTR% or GACTF% in the TOPS-20 Monitor Calls Reference Manual); and some calls do not return at all (see HALTF% or WAIT%). Refer to Chapter 3 of the TOPS-20 Monitor Calls Reference Manual for the possible returns for each monitor call.

When a failure occurs during the execution of a monitor call, the monitor stores an error code. The error code indicates the cause of the failure. This error code is usually stored in the right half of AC1, but can also be stored in the monitor's data base or a user's data block. In either case, you can obtain the message associated with the error by using the GETER% or ERSTR% call.

The ERJMP/ERCAL facilities can also be used following a machine instruction, and will trap for the following conditions:

- o Illegal instruction
- o Illegal memory read
- o Illegal memory write
- o Pushdown list overflow

The ERJMP/ERCAL facilities can be used after all monitor calls, regardless of whether the call has one or two returns. To handle errors consistently, users are encouraged to employ either the ERJMPR, ERCALR, ERJMPS, or ERCALS symbol with all calls. All of the six predefined jump symbols are no-ops, unless they immediately follow a monitor call or instruction that fails. Error codes can be obtained by the program and translated into their corresponding error mnemonic and message strings with the GETER% and ERSTR% monitor calls.

TOPS-20 provides convenient macros and subroutines for handling monitor call error routines. They can be found in the system file MACSYM.MAC. Two such macros are EJSERR and EJSHLT. EJSERR prints out an error message and returns control to the next instruction following the failing monitor call. EJSHLT prints out an error message and halts processing of the program.

The following is an example of executing the BIN% monitor call (see Chapter 3 for more information on this monitor call) that has a single

return. If the execution of the call is successful, the program reads and stores a character. If the execution of the call is not successful, the program transfers control to an error routine. This routine processes the error and then returns control back to the main program sequence. Note that ERCALS stores the return address on the stack.

```
DOIT:   MOVE    T1,INJFN     ;obtain JFN for input file
        BIN%                 ;input one character
         ERCALS ERROR        ;call error routine if problem
        MOVEM   T2,CHAR      ;store character
        JRST    DOIT         ;and get another
ERROR:  GTSTS%               ;read file status
        TXNE    T2,GS%EOF    ;end of file?
        JRST    EOF          ;yes, process end-of-file condition
        HRROI   T1,[ASCIZ/
?INPUT ERROR, CONTINUING
/]                     ;no, data error
        PSOUT%               ;print message
        RET                  ;return to program (POPJ 17,)
```

The ASCIZ pesudo-op specifies a left-justified ASCII string terminated with a null (that is, a byte containing all bits equal to zero) byte.

1.3 PROGRAM ENVIRONMENT

The user program environment in the TOPS-20 operating system consists of a job structure that can contain many processes. A process is a runnable or schedulable entity capable of performing computations in parallel with other processes. This means that a runnable program is associated with at least one process.

Each process has its own address space for storing its computations. This address space is called virtual space because it is actually a "window" into physical storage. The address space is divided into 32 (decimal) sections. Each section is divided into 512 (decimal) pages, and each page contains 512 (decimal) words. Each word contains 36 bits.

A process can communicate with other processes in the following ways:

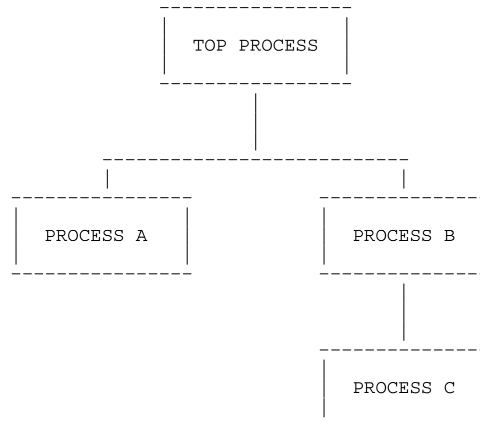
- o explicitly, by software interrupts or system facilities (the inter-process communication facility, or IPCF, for example).
- o implicitly, by changing parts of its environment (its address space, for instance) that are being shared with other processes.

A process can create other processes inferior to it, but there is one control process from which the chain of creations begins. A process

is said to exist when a superior process creates it and is said to end when a superior process deletes it. Refer to Chapter 5 for more information on the process structure.

A set of one or more related processes, normally under control of a single user, is a job. Each active process is part of some job on the system. A job is defined by a user name, an account number, some open files, and a set of running and/or suspended processes. A job can be composed of several running or suspended programs.

The following diagram illustrates a job structure consisting of four processes.



Both process A and 1 process B are created by the TOP PROCESS and thus are directly inferior to it. Process C is created by process B and thus is directly inferior to process B only. Process C is indirectly inferior to the TOP PROCESS.

In summary, processes can be considered as independent virtual jobs with well-defined relationships to other processes in the system, and a job is a collection of these processes.

CHAPTER 2

INPUT AND OUTPUT USING THE TERMINAL

One of the main reasons for using monitor calls is to transfer data from one location to another. This chapter discusses moving data to and from the user's terminal.

2.1 OVERVIEW

Data transfers to and from the terminal are in the form of either individual bytes or text strings. The bytes are 7-bit bytes. The strings are ASCII strings ending with a 0 byte. These strings are called ASCIZ strings.

To designate the desired string, the user's program must include a statement that points to the beginning of the string being read or written. The MACRO pseudo-op, POINT, can be used to set up this pointer, as shown in the following sequence of statements:

```
            MOVE AC1,PTR
                 .
                 .
                 .
     PTR:   POINT 7,MSG
     MSG:   ASCIZ/TEXT MESSAGE/
```

Accumulator 1 contains the symbolic address (PTR) of the pointer. At the address specified by PTR is the pointer to the beginning of the string. The pointer is set up by the POINT pseudo-op. The general format of the POINT pseudo-op is:

```
     POINT decimal-byte-size,address,decimal-byte-position
```

(Refer to the _TOPS-20 MACRO Assembler Reference Manual_ for more information on the POINT pseudo-op.) In the example above, the POINT pseudo-op has been written to indicate 7-bit bytes starting before the left-most bit in the address specified by MSG.

Another way of setting up an accumulator to contain the address of the pointer is with the following statement:

```
     HRROI AC1,[ASCIZ/TEXT MESSAGE/]
```

The instruction mnemonic HRROI causes a -1 to be placed in the left half of accumulator 1 and the address of the string to be placed in the right half. However, in the above statement, a literal (enclosed in square brackets) has been used instead of a symbolic address. The literal causes the MACRO assembler to:

- o store data within brackets (the string) in a table.
- o assign an address to the first word of the data.
- o insert that address as the operand to the HRROI instruction.

Literals have the advantage of showing the data at the point in the program where it will be used, instead of showing it at the end of the program.

As far as the I/O monitor calls are concerned, a word in this format (-1 in the left half and an address in the right half) designates the system's standard pointer (that is, a pointer to a 7-bit ASCIZ string beginning before the leftmost byte of the string). The result of the HRROI statement is interpreted by the monitor as functionally equivalent to the word assembled by the POINT 7, address pseudo-op and is the recommended statement to use in preparation for a monitor call. However, byte manipulation instructions (for example, ILDB, IBP, ADJBP) will not operate properly with this type of pointer.

After a string is read, the pointer is advanced to the character following the terminating character of the string. After a string is written, the pointer is advanced to the character following the last non-null character written.

Most TOPS-20 monitor calls accept one-word global byte pointers when executed from a nonzero section (see Section 8.3). Global byte pointers are used with extended addressing and are fully explained in Chapter 8 of this document. Unless specifically stated, TOPS-20 monitor calls do not accept two-word global byte pointers.

2.2 PRIMARY I/O DESIGNATORS

To transfer data from one location to another, the user's program must indicate the source from which the data is to be obtained and the destination where the data is to be placed. By default, the user's terminal is defined as the source and destination. The default can be overridden by using the SPJFN% monitor call (refer to the _TOPS-20 Monitor Calls Reference Manual_). Examples in this book assume the

user's terminal to be the source (input) and destination (output) device. Two designators are used to represent the user's terminal:

1. The symbol .PRIIN to represent the user's terminal as the source (input) device.

2. The symbol .PRIOU to represent the user's terminal as the destination (output) device.

These symbols are called the primary input and output designators and by default are used to represent the terminal running the program. They are defined in the system file MONSYM.MAC and do not have to be defined in the user's program as long as the program contains the statement

```
SEARCH MONSYM
```

2.3 PRINTING A STRING

Many times a program may need to print an error message or some other string, such as a prompt to request input from the user at the terminal. The PSOUT% (Primary String Output) monitor call is used to print such a string on the terminal. This call copies the designated string from the program's address space. Thus, the source of the data is the program's address space, and the destination for the data is the terminal. The program need only supply the pointer to the string being printed.

Accumulator 1 (AC1) is used to contain the address of the pointer. After AC1 is set up with the pointer to the string, the next line of code is the PSOUT% call. Thus, an example of the PSOUT% call is:

```
HRROI AC1,[ASCIZ/TEXT MESSAGE/]    ;string to print
PSOUT%                             ;print TEXT MESSAGE
```

The PSOUT% call prints on the terminal all the characters in the string until it encounters a null byte. Note that the string is printed exactly as it is stored in the program, starting at the current position of the terminal's print head or cursor and ending with the last character in the string. If a carriage return and line feed are to be output, either before or after the string, these characters should be inserted as part of the string. For example, to print TEXT MESSAGE on one line and to output a carriage return-line feed after it, the user's program includes the call

```
     HRROI AC1,[ASCIZ/TEXT MESSAGE
/]
     PSOUT%
```


After the string is printed, the instruction following the PSOUT% call in the user's program is executed. Also, the pointer in AC1 is updated to point to the character following the last non-null character written.

The macro TMSG, found in the system file MACSYM, does the same thing as the example above. This macro offers the programmer a convenient way for printing messages on the terminal. For example

```
TMSG <TEXT MESSAGE
>
```

caused the text message contained between the angle brackets, including the carriage return and line feed, to print on the terminal. The TMSG macro, along with others previously mentioned, will be used consistently in examples throughout this document. Refer to the system file MACSYM.MAC for further information on MACSYM macros.

Refer to Section 1.2.2 for information concerning error returns.

2.4 READING A NUMBER

The NIN% (Number Input) monitor call is used to read an integer. This call does not assume the terminal as the source designator; therefore, the user's program must specify this. The NIN% call accepts the number from any valid source designator, including a string in memory. This section discusses reading a number directly from the terminal. Refer to Section 2.9 for an example of using the NIN% call to read the number from a string in memory. The destination for the number is AC2, and the NIN% call places the binary value of the number read into this accumulator. The user's program also specifies a number in AC3 that represents the radix of the number being input. The radix given can be in the range 2-36.

Thus, the setup for the NIN% monitor call is the following:

```
MOVEI AC1,.PRIIN    ;AC1 contains the primary input designator
                    ;(the user's terminal)

MOVEI AC3,^D10      ;AC3 contains the radix of the number being
                    ;input (in this case a decimal number)

NIN%                ;The call to input the number
```

After completion of the NIN% call, control returns to the program at one of two places (refer to Section 1.2.2). If an error occurs during the execution of the call, control returns to the instruction following the call. This instruction should be a jump-type instruction to an error processing routine (see Section 1.2.2). Also, an error code is placed in AC3 (refer to Appendix B of the _TOPS-20 Monitor Calls Reference Manual_ for the error codes). If the execution

of the NIN% call is successful, control returns to the second instruction following the call. The number input from the terminal is placed in AC2.

The NIN% call terminates when it encounters a nondigit character (for example, a letter, a punctuation character, or a control character). This means that if 32X1 were typed on the terminal, on return AC2 contains a 40 (octal) because the NIN% call terminated when it read the X.

The following program prints a message and then accepts a decimal number from the user at the terminal. Note that the NIN% call terminates reading on any nondigit character; therefore, the user cannot edit his input with any of the editing characters (for example, DELETE, CTRL/W). The RDTTY% call (refer to Section 2.9) should be used in programs that read from the terminal because it allows the user to edit his input as he is typing it.

```
       SEARCH MONSYM
       HRROI AC1,[ASCIZ/
Enter # of seconds:  /]
       PSOUT%                ;output a prompt message
       MOVEI AC1,.PRIIN      ;input from the terminal
       MOVEI AC3,^D10        ;use the decimal radix
       NIN%                  ;input a decimal number
        ERJMP NINERR         ;error-go to error routine
       MOVEM AC2, NUMSEC     ;save number entered
         .
         .
         .
NUMSEC:BLOCK 1
         .
         .
         .
```

2.5 WRITING A NUMBER

The NOUT% (Number Output) monitor call is used to output an integer. The user's program moves the number to be output into AC2. The program must specify the destination for the number in AC1 and the radix in which the number is to be output in AC3. The radix given cannot be greater than base 36. In addition, the user's program can specify certain formatting options to be used when printing the number.

Thus, the general setup for the NOUT% monitor call is as follows:

```
     AC1:   output designator
     AC2:   number being output
     AC3:   format options in left half and radix in right half
```


The format options that can be specified in the left half of AC3 are described in Table 2-1.

Table 2-1: NOUT% Format Option

Bit	Symbol	Meaning
0	NO%MAG	Print the number as a positive 36-bit number. For example, -1 would be printed as 777777 777777 if radix=8).
1	NO%SGN	Print the appropriate sign (+ or -) before the number. If bits NO%MAG and NO%SGN are both on, a plus sign is always printed.
2	NO%LFL	Print leading filler. If this bit is not set, trailing filler is printed and bit NO%ZRO is ignored.
3	NO%ZRO	Use 0's as the leading filler if the specified number of columns allows filling. If this bit is not set, blanks are used as the leading filler if the number of columns allows filling.
4	NO%OOV	Output on column overflow and return an error. If this bit is not set, column overflow is not output.
5	NO%AST	Print asterisks when the column overflows. If this bit is not set, and bit 4 (NO%OOV) is set, all necessary digits are printed when the columns overflow.
6-10		Reserved for Digital (must be 0).
11-17	NO%COL	Print the number of columns indicated. This value includes the sign column. If this field is 0, as many columns as necessary are printed.

The following instruction sequence is an example of the NOUT% monitor call. This sequence prints a number, stored in location NUMB, on the user's terminal. The number can be positive, negative or zero, with no special formatting.

```
        MOVX AC1,.PRIOU                 ;use primary output
        MOVE AC2,NUMB                   ;get number from location NUMB
        MOVX AC3,^D10                   ;no special format
                                        ;decimal radix
        NOUT%                           ;print number
         EJSHLT                         ;unexpected fatal error. Halt
                                        ;and print message.
```

Refer to Section 1.2.2 for information concerning error returns. The following example illustrates the use of the three monitor calls described so far, as well as the TMSG macro. The RESET% and HALTF% monitor calls are described in Section 2.6.

```
        SEARCH MONSYM
        SEARCH MACSYM
        .REQUIRE SYS:MACREL
        AC1==1
        AC2==2
        AC3==3
START:  RESET%                          ;prepare program environment
        HRROI AC1,[ASCIZ/PLEASE TYPE A DECIMAL NUMBER: /]
        PSOUT%
        MOVEI AC1,.PRIIN                ;source designator
        MOVEI AC3,^D10                  ;decimal radix
        NIN%
         ERJMPS ERROR                   ;if input error print message
                                        ;halt.
        TMSG <THE OCTAL EQUIVALENT IS >
        MOVEI AC1,.PRIOU                ;destination designator
        MOVEI AC3,^D8                   ;octal radix
        NOUT%
         EJSHLT                         ;fatal error.
                                        ;Same as ERJMPS ERROR.
        HALTF%                          ;return to command language
        JRST START                      ;begin again, if continued
ERROR:  TMSG<
?ERROR-TYPE START TO BEGIN AGAIN>
        HALTF%
        JRST START                      ;user types continue-start
                                        ;again
        END START
```

2.6 INITIALIZING AND TERMINATING THE PROGRAM

Two monitor calls that have not yet been described were used in the above program - RESET% and HALTF%.

2.6.1 RESET% Monitor Call

A good programming practice is to include the RESET% monitor call at the beginning of every assembly language program. This call closes any existing open files and releases their JFNs, kills any inferior processes, clears the software interrupt system (see Chapter 4), and performs various other process initilization functions. For a complete list of the functions provided by the RESET% monitor call, refer to the description of the call in the TOPS-20 Monitor Calls Reference Manual. The format of the call is

```
     RESET%
```

and control always returns to the next instruction following the call.

2.6.2 HALTF% Monitor Call

To stop the execution of a program and return control to the TOPS-20 Command Language, the user must include the HALTF% monitor call as the last instruction performed in the program. The user can then resume execution of the program at the instruction following the HALTF% call by typing the CONTINUE command after control has returned to command level.

2.7 READING A BYTE

The PBIN% (Primary Byte Input) monitor call is used to read a single byte (that is, one character) from the terminal. The user's program does not have to specify the source and destination for the byte because this call uses the primary input designator (that is, the user's terminal) as the source and accumulator 1 as the destination. After execution of the PBIN% call, control returns to the instruction following the PBIN%. If execution of the call is successful, the byte read from the terminal is right-justified in AC1. If execution of the call is not successful, an illegal instruction trap is generated, as explained in Section 1.2.2.

2.8 WRITING A BYTE

The PBOUT% (Primary Byte Output) monitor call is used to write a single byte to the terminal. This call uses the primary output designator (that is, the user's terminal) as the destination for the byte; thus, the user's program does not have to specify the destination. The source of the byte being written is accumulator 1; therefore, the user's program must place the byte right-justified in AC1 before the call.

After execution of the PBOUT% call, control returns to the instruction following the PBOUT%. If execution of the call is successful, the byte is written to the user's terminal. If execution of the call is not successful, an illegal instruction trap is generated, as explained in Section 1.2.2.

2.9 READING A STRING

Up to this point, monitor calls have been presented for printing a string, reading and writing an integer, and reading and writing a byte. The next call to be discussed obtains a string from the terminal and, in addition, allows the user at the terminal to edit his input as he is typing it.

The RDTTY% (Read from Terminal) monitor call reads input from the user's terminal (that is, from .PRIIN) into the program's address space. Input is read until the user either types an appropriate terminating (break) character or inputs the maximum number of characters allowed in the string, whichever occurs first. Output generated as a result of character editing is printed on the user's terminal (that is, output to .PRIOU).

The RDTTY% call handles the following editing functions:

1. Delete the last character in the string if the user presses the DELETE key while typing his input.
2. Delete back to the last punctuation character in the string if the user types CTRL/W while typing his input.
3. Delete the current line if the user types CTRL/U while typing his input.
4. Retype the current line if the user types CTRL/R while typing his input.

Because the RDTTY% call can handle these editing functions, a program can accept input from the terminal and allow this input to be corrected by the user as he is typing it. For this reason, the RDTTY call should be used to read input from the terminal before processing that input with calls such as NIN%.

The RDTTY% call accepts three words of arguments in AC1 through AC3.

AC1: pointer to area in program's address space where input is to be placed. This area is called the text input buffer.

AC2: control bits in the left half, and maximum number of bytes in the text input buffer in the right half.

AC3: pointer to buffer for text to be output before the user's input if the user types a CTRL/R, or 0 if only the user's input is to be output on a CTRL/R.

The control bits in the left half of AC2 specify the characters on which to terminate the input. These bits are described in Table 2-2.

Table 2-2: RDTTY% Control Bits

Bit	Symbol	Meaning
0	RD%BRK	Terminate input when user types a CTRL/Z or presses the ESC key.
1	RD%TOP	Terminate input when user types one of the following: CTRL/G CTRL/L CTRL/Z ESC key RETURN key Line feed key
2	RD%PUN	Terminate input when user types one of the following: CTRL/A-CTRL/F CTRL/H-CTRL/I CTRL/K CTRL/N-CTRL/Q CTRL/S-CTRL/T CTRL/X-CTRL/Y ASCII codes 34-36 ASCII codes 40-57 ASCII codes 72-100 ASCII codes 133-140 ASCII codes 173-176 The ASCII codes listed above represent the punctuation characters in the ASCII character set. Refer to the ASCII character set table in Appendix A of the _TOPS-20 Monitor Calls Reference Manual_ for these characters.
3	RD%BEL	Terminate input when user types the RETURN or line feed key (that is, end of line).

4	RD%CRF	Store only the line feed in the input buffer when the user presses the RETURN key. A carriage return will still be output to the terminal but will not be stored in the buffer. If this bit is not set and the user presses the RETURN key, both the carriage return and the line feed will be stored as part of the input.
5	RD%RND	Return to program if the user attempts to delete past the beginning of his input. This allows the program to take control if the user tries to delete all of his input. If this bit is not set, the program waits for more input.
6		Reserved for Digital (must be 0).
7	RD%RIE	Return to program when there is no input (that is, the text input buffer is empty). If this bit is not set, the program waits for more input.
8		Reserved for Digital (must be 0).
9	RD%BEG	Return to user program if the user attempts to edit beyond the beginning of the input buffer.
10	RD%RAI	Convert lower case input to upper case.
11	RD%SUI	Suppress the CTRL/U indication on the terminal when a CTRL/U is typed by the user. This means that if the user types a CTRL/U, XXX will not be printed and, on display terminals, the characters will not be deleted from the screen. If this bit is not set and the user types a CTRL/U, XXX will be printed and, if appropriate, the characters will be deleted from the screen. In neither case is the CTRL/U stored in the input buffer.

15	RD%NED	Disable editing characters in user break mask. If this bit is set, then any editing character (^R, ^U, ^V, ^W, and DELETE) in the user supplied break mask does not have its editing function.

If no control bits are set in the left half of AC2, the input will be terminated when the user presses the RETURN or line feed key (that is, terminated on an end-of-line condition only).

The count in the right half of AC2 specifies the number of bytes available for storing the string in the program's address space. The input is terminated when this count is exhausted, even if a specified break character has not yet been typed.

The pointer in AC3 is to the beginning of a buffer containing the text to be output if the user types a CTRL/R. When this happens, the text in this separate buffer is output, followed by any text that has been typed by the user. The text in this buffer cannot be edited with any of the editing characters (that is, DELETE, CTRL/W, or CTRL/U). If the contents of AC3 is zero, then no such buffer exists, and if the user types CTRL/R, only the text in the input buffer will be output.

If execution of the RDTTY% call is successful, the input is in the specified area in the program's address space. The character that terminated the input is also stored. (If the terminating character is a carriage return followed by a line feed, the line feed is also stored.) Control returns to the user's program at the second location following the call. The pointer in AC1 is advanced to the character following the last character stored. The count in the right half of AC2 is updated to reflect the remaining bytes in the buffer, and appropriate bits are set in the left half of AC2. The bits that can be set on a successful return are:

Bit 12	RD%BTM	The input was terminated because one of the specified break characters was typed. This break character is placed in the input buffer. If this bit is not set, the input was terminated because the byte count was exhausted.
Bit 13	RD%BFE	Control was returned to the program because there is no more input and RD%RIE was set in the call.

Bit 14 RD%BLR The limit to which the user can backup for editing his input was reached.

For consistent handling of error returns refer to Section 1.2.2.

The following example illustrates the recommended method for reading data from the terminal. This example is essentially the same as the one in Section 2.5; however, the RDTTY% call is used to read the number before the NIN% call processes it. This program stores the last error encountered in location LASTER and therefore uses the ERJMPR pseudo-op.

```
        SEARCH MONSYM
        SEARCH MACSYM
        .REQUIRE SYS:MACREL
        AC1==1
        AC2==2
        AC3==3
START:  RESET%                  ;prepare program environment
        HRROI AC1,PROMPT
        PSOUT%                  ;type prompt
        HRROI AC1,BUFFER        ;location to store number
        MOVEI AC2,BUFLEN*5      ;size of buffer
        HRROI AC3,PROMPT        ;pointer to prompt
        RDTTY%                  ;read number from term. with editing
         ERJMPR ERROR           ;save error code, print message
        HRROI AC1,BUFFER        ;and halt source designator
        MOVEI AC3,^D10          ;decimal radix
        NIN%
         ERJMPR ERROR           ;if input error, print message
        TMSG <THE OCTAL EQUIVALENT IS >
        MOVEI AC1,.PRIOU        ;and halt destination designator
        MOVEI AC3,^D8           ;octal radix
        NOUT%
         ERJMPR ERROR           ;save error code, print message
        HALTF%                  ;and halt return to command
        JRST START              ;language begin again, if continued
PROMPT: ASCIZ/PLEASE TYPE A DECIMAL NUMBER: /
        BUFLEN==10
BUFFER: BLOCK BUFLEN
LASTER: BLOCK 1
ERROR:  MOVEM AC1,LASTER        ;save error code
        TMSG <
?ERROR-TYPE START TO BEGIN AGAIN>;print general error message
        HALTF%                  ;halt
        JRST START              ;start over if continued
        END START
```


2.10 SUMMARY

Data transfers of sequential bytes or text strings can be made to and from the terminal. The monitor calls for transferring bytes are PBIN% and PBOUT% and for transferring strings are PSOUT% and RDTTY%. The NIN% and NOUT% monitor calls can be used for reading and writing a number. In general, the user's program must specify a source from which the data is to be obtained and a destination where the data is to be placed. In the case of terminal I/O, the symbol .PRIIN represents the user's terminal as the source, and the symbol .PRIOU represents the user's terminal as the destination.

CHAPTER 3

USING FILES

3.1 OVERVIEW

All information stored in the DECSYSTEM-20 is kept in files. The basic unit of storage in a file is a page containing bytes from 1 to 36 bits in length. Thus, a sequence of pages constitutes a file. In most cases, files have names. Although all files are handled in the same manner, certain operations are unavailable for files on particular devices.

Programs can reference files by several methods:

- In a sequential byte-by-byte manner.
- In a multiple byte or string manner.
- In a random byte-by-byte manner if the particular file-storage device allows it.
- In a page-mapping or section-mapping manner for files on disk.

Byte and string input/output are the most common types of operations.

Generally, all programs perform I/O by moving bytes of data from one location to another. For example, programs can move bytes from one memory area to another, from memory to a disk file, and from the user's terminal to memory. In addition, a program can map multiple 512-word pages or 512-page sections from a disk file into memory or vice versa.

Data transfer operations on files require four steps:

1. Establishing a correspondence between a file and a Job File Number (JFN), because all files are referenced by JFNs.
2. Opening the file to establish the data mode, access mode, and byte size and to set up the monitor tables that permit data to be accessed.

3. Transferring data either to or from the file.
4. Closing the file to complete any I/O, to update the directory if the file is on the disk, and to release the monitor table space used by the file.

Some operations on files do not require the execution of all four steps above. Examples of these operations are: deleting or renaming a file, or changing the access code or account of a file. Although these operations do not require all four steps, they do require that the file has a JFN associated with it (step 1 above).

It is possible for disk files on the DECSYSTEM-20 to be simultaneously read or written by any number of processes. To make sharing of files possible, all instances of opening a specific file in a specific directory cause a reference to the same data. Therefore, data written into a file by one process can immediately be seen by other processes reading the file.

Access to files is controlled by the 6-digit (octal) file access code assigned to a file when it is created. This code indicates the types of access allowed to the file for the three classes of users: the owner of the file, the users with group access to the file, and all other users. (Refer to the TOPS-20 User's Guide for more information on the file access codes.) If the user is allowed access to a file, he requests the type of access desired when opening the file with the OPENF% monitor call (refer to Section 3.4.1) in his program. If the access requested in the OPENF% call does not conflict with the current access to the file, the user is granted access. Essentially, the current access to the file is set by the first user who opens it.

Thus, for a user to be granted access to a specific file, two conditions must be met:

1. The file access code must allow the user to access the file in the desired manner (for example, read, write).
2. The file must not be opened for a conflicting type of access.

3.2 JOB FILE NUMBER

The Job File Number (JFN) is one of the more important concepts in the operating system because it serves as the identifier of a particular file on a particular device during a process' execution. It is a small integer assigned by the system upon a request from the user's program. JFNs are usually assigned sequentially starting with 1.

The JFN is valid for the job in which it is assigned and may be used by any process in the job. The system uses the JFN as an index into the table of files associated with the job and always assigns a JFN that is unique within the job. Even though a particular JFN within the job can refer to only one file, a single file can be associated with more than one JFN. This occurs when two or more processes are using the same file concurrently. In this case, each of the processes will probably have a different JFN for the file, but all of the JFNs will be associated with the same file.

3.3 ASSOCIATING A FILE WITH A JFN

In order to reference a file, the first step the user program must complete is to associate the specific file with a JFN. This correspondence is established with the GTJFN% (Get Job File Number) monitor call. One of the arguments to this call is the string representing the desired file. The string can be specified within the program (that is, come from memory) or can be accepted as input from the user's terminal or from another file. The string can represent the complete specification for the file:

```
dev:<directory>name.typ.gen;T(temporary);P(protection);A(account);
     (device dependent attributes)
```

If you omit any fields of the specification, the system can provide values for all except the name field. Refer to the TOPS-20 User's Guide for a complete explanation of the specification for a file.

Table 3-1 lists the values the system will assign to fields not specified by the input string.

Table 3-1: Standard System Values for File Specifications

Field	Value
Device	DSK:
Directory	Directory to which user is currently connected.
Name	No default; this field must be specified.
Type	Null.
Generation number	The highest existing generation number if the file is an input file. The next higher generation number if the file is an output file.
Protection	Protection of next lower generation of file, if one exists; otherwise, protection as specified in the directory.
Account	Account specified when user logged in.

If the string specified identifies a single file, the monitor returns a JFN that remains associated with that file until either the process releases the JFN or the job logs off the system. After the assignment of the JFN is complete, the user's program uses the JFN in all references to that file.

The user's program can set up either the short or the long form of the GTJFN% monitor call. The long form of the GTJFN% call requires an argument block; the short form does not. The long form of GTJFN% has functions and flexibility not available in the short form of the call. The short form of GTJFN% allows a file specification to be obtained from a string in memory or from a file, but not from both. Fields not specified by the input are taken from the standard system values for those fields (refer to Table 3-1). This form is sufficient for most uses of the call. The long form allows a file specification to be obtained from both a string in memory and a file. If both are given as arguments, the string is used first, and then the file is used if more fields are needed to complete the specification. This form also allows the user's program to specify nonstandard values to be used for fields not given and to request the assignment of a specific JFN.

3.3.1 GTJFN% Monitor Call

The GTJFN% monitor call assigns a JFN to the specified file. It accepts two words of arguments. These argument words are different depending on the form of GTJFN% being used. The user's program indicates the desired GTJFN% form by setting bit 17(GJ%SHT) of AC1 to 1 for the short form or by clearing bit 17(GJ%SHT) for the long form.

3.3.1.1 Short Form of GTJFN% - The short form of the GTJFN% monitor call requires the following two words of arguments.

```
         0                        17 18                          35
        !=======================================================!
 AC1    !         flag bits         ! default generation number !
        !=======================================================!

         0                                                       35
        !=======================================================!
 AC2    !     source designator for file specification per      !
        !          bit 16  (GJ%FNS) of AC1                      !
        !=======================================================!
```

The flag bits that can be specified in AC1 are described in Table 3-2.

Table 3-2: GTJFN% Flag Bits

Bit	Symbol	Meaning
0	GJ%FOU	The file specification given is to be assigned the next higher generation number. This bit indicates that a new version of a file is to be created and is normally set if the file is for output use.
1	GJ%NEW	The file specification given must not refer to an existing file (that is, the file must be a new file).
2	GJ%OLD	The file specification given must refer to an existing file. This bit has no effect on a parse-only JFN. (See bit GJ%OFG.)
3	GJ%MSG	One of the appropriate messages is to be printed after the file specification is obtained. The message is printed only if the user types the ESC key to end his file specification (that is, he is using recognition input). [NEW FILE] [NEW GENERATION] [OLD GENERATION] [OK] if GJ%CFM (bit 4) is off [CONFIRM] if GJ%CFM (bit 4) is on

Bit	Symbol	Meaning
4	GJ%CFM	Confirmation from the user will be required to verify that the file specification obtained is correct. To confirm the file specification, the user can press the RETURN key.
5	GJ%TMP	The file specified is to be a temporary file.
6	GJ%NS	Only the first file specification in a multiple logical name assignment is to be searched for the file.
7	GJ%ACC	The JFN specified is not to be accessed by inferior processes in this job. However, any process can access the file by acquiring a different JFN. To prevent the file from being accessed by other processes, the user's program can set OF%RTD (bit 29) in the OPENF call (refer to Section 3.4.1).
8	GJ%DEL	The file specified is not to be considered as deleted, even if it is marked as deleted.
9-10	GJ%JFN	These bits are off in the short form of the GTJFN call (refer to Section 3.3.1.2 for their description).
11	GJ%IFG	The file specification given is allowed to have one or more of its fields specified with a wildcard character (* or %). This bit is used to process a group of files and is generally used for input files. The monitor verifies that at least one value exists for each field that contains a wildcard and assigns the JFN to the first file in the group. The monitor also verifies that fields not containing wildcards represent a new or old file according to the setting of GJ%NEW and GJ%OLD.
12	GJ%OFG	The JFN is to be associated with the given file specification string only and not to the actual file. The string may contain a wildcard character (* or %) in one or more of

		its fields. It is checked for correct punctuation between fields, but is not checked for the validity of any field. This bit allows a JFN to be associated with a file specification even if the file specification does not refer to an actual file. The JFN returned cannot be used to refer to an actual file (for example, cannot be used in an OPENF call) but can be used to obtain the original input string via the JFNS monitor call (refer to Section 3.7.2).
13	GJ%FLG	Flags are to be returned in the left half of AC1 on a successful return.
14	GJ%PHY	Logical names specified for the current job are to be ignored and the physical device is to be used.
15	GJ%XTN	This bit is off in the short form of the GTJFN call (refer to Section 3.3.1.2 for its description).
16	GJ%FNS	The contents of AC2 are to be interpreted as follows: 1. If this bit is on, AC2 contains an input JFN in the left half and an output JFN in the right half. The input JFN is used to obtain the file specification to be associated with the JFN. The output JFN is used to indicate the destination for printing the names of any fields being recognized. To omit either JFN, the user's program must specify the symbol .NULIO (377777). 2. If this bit is off, AC2 contains a pointer to a string in memory that specifies the file to be associated with the JFN.
17	GJ%SHT	This bit must be on (set) for the short form of the GTJFN% call; it must be off for the long form of the call.

18-35		The generation number of the file (between 1 and 377777) or one of the following: 0(.GJDEF) to indicate that the next higher generation number of the file is to be used if GJ%FOU (bit 0) is on, or to indicate that the highest existing generation number of the file is to be used if GJ%FOU is off. (This value is usually used in this field.) -1(.GJNHG) to indicate that the next higher generation number of the file is to be used if no generation number is supplied. -2(.GJLEG) to indicate that the lowest existing generation number of the file is to be used. -3(.GJALL) to indicate that all generation numbers (*) of the file are to be used and that the JFN is to be assigned to the first file in the group. (Bit GJ%IFG must be set.)

If the GTJFN% call is given with the appropriate flag bit set (GJ%IFG or GJ%OFG), the file specification given as input can have a wildcard character (either an asterisk or a percent sign) appearing in the directory, name, type, or generation number field. (The percent sign cannot appear in the generation number field.) The wildcard character is interpreted as matching any existing occurrence of the field. For example, the specification

<LIBRARY>*.MAC

identifies all the files with the file type .MAC in the directory named <LIBRARY>. The specification

<LIBRARY>MYFILE.FO%

identifies all the files in directory <LIBRARY> with the name MYFILE and a three-character file type in which the first two characters are .FO. Upon completion of the GTJFN call, the JFN returned is associated with the first file found in the group according to the following:

- in numerical order by directory number
- in alphabetical order by filename
- in alphabetical order by file type
- in ascending numerical order by generation number

The GNJFN% (Get Next JFN) monitor call can then be given to assign the JFN to the next file in the group (refer to Section 3.7.3). Normally, a program that accepts wildcard characters in a file specification will successively reference all files in the group using the same JFN and not obtain another JFN for each one.

If execution of the GTJFN% call is not successful because problems were encountered in performing the call, the JFN is not assigned and an error code is returned in the right half of AC1. The execution of the program continues at the instruction following the GTJFN% call.

If execution of the GTJFN% call is successful, the JFN assigned is returned in the right half of AC1 and various bits are set in the left half, if flag bits 11, 12, or 13 were on in the call. (The bits returned on a successful call are described in Table 3-3.) If bit 11, 12, or 13 was not on in the call, the left half of AC1 is zero. The execution of the program continues at the second instruction after the GTJFN% call.

Table 3-3: Bits Returned on GTJFN% Call

Bit	Symbol	Meaning
0	GJ%DEV	The device field of the file specification contains wildcard characters.
1	GJ%UNT	The unit field of the file specifications contains wildcard characters. This bit is never set because wildcard characters are not allowed in unit fields.
2	GJ%DIR	The directory field of the file specification contains wildcard characters.
3	GJ%NAM	The filename field of the file specification contains wildcard characters.
4	GJ%EXT	The file type field of the file specification contains wildcard characters.
5	GJ%VER	The generation number field of the file specification contains wildcard characters.
6	GJ%UHV	The file used has the highest generation number because a generation number of 0 was given in the call.
7	GJ%NHV	The file used has the next higher generation number because a generation number of 0 or -1 was given in the call.
8	GJ%ULV	The file used has the lowest generation number because a generation number of -2 was given in the call.
9	GJ%PRO	The protection field of the file specification was given.
10	GJ%ACT	The account field of the file specification was given.

11	GJ%TFS	The file specification is for a temporary file.
12	GJ%GND	Files marked for deletion are not considered when assigning JFNs in subsequent calls. This bit is set if GJ%DEL was not set in the call.
13	GJ%NOD	The node name field of the file specification was given.
17	GJ%GIV	Invisible files were not considered when assigning JFNs.

Examples of the short form of the GTJFN% monitor call are shown in the following paragraphs.

The following sequence of instructions is used to obtain, from the user's terminal, the specification of an existing file.

```
     MOVX AC1,GJ%OLD+GJ%FNS+GJ%SHT
     MOVE AC2,[.PRIIN,,.PRIOU]
     GTJFN%
```

The bits specified for AC1 indicate that the file specification given must refer to an existing file (GJ%OLD), that the file specification is to be accepted from the input JFN in AC2 (GJ%FNS), and that the short form of the GTJFN% call is being used (GJ%SHT). Because the right half of AC1 is zero, the standard generation number algorithm will be used. In this GTJFN% call, the file with the highest existing generation number is used. Because GJ%FNS is set in AC1, the contents of AC2 are interpreted as containing an input JFN and an output JFN. In this example, the file specification is obtained from the terminal (.PRIIN).

The following sequence of instructions is used to obtain, from the user's terminal, the specification of an output file and to require confirmation from the user once the file specification has been obtained.

```
     MOVX AC1,GJ%FOU+GJ%MSG+GJ%CFM+GJ%FNS+GJ%SHT
     MOVE AC2,[.PRIIN,,.PRIOU]
     GTJFN%
```

In this example, the bits specified for AC1 indicate that

- the file obtained is to be an output file (GJ%FOU),
- after the file specification is obtained, a message is to be typed (GJ%MSG),

- the user is required to confirm the file specification that was obtained (GJ%CFM),
- the file specification is to be obtained from the input JFN in AC2 (GJ%FNS),
- the short form of the GTJFN% call is being used (GJ%SHT).

Because the right half of AC1 is zero, the generation number given to the file will be one greater than the highest generation number existing for the file. The contents of AC2 are interpreted as containing an input JFN and an output JFN because GJ%FNS is set in AC1.

The following sequence of instructions is used to obtain the name of an existing file from a location in the user's program.

```
     MOVX AC1,GJ%OLD+GJ%SHT
     MOVE AC2,[POINT 7,NAME]
     GTJFN%
        .
        .
        .

NAME:ASCIZ/MYFILE.TXT/
```

The bits specified for AC1 indicate that the file obtained is to be an existing file (GJ%OLD) and that the short form of the GTJFN% call is being used (GJ%SHT). Since the right half of AC1 is zero, the file with the highest generation number will be used. Because GJ%FNS is not set, the contents of AC2 are interpreted as containing a pointer to a string in memory that specifies the file to be associated with the JFN. The setup of AC2 indicates that the string begins at location NAME in the user's program. The file specification obtained from location NAME is MYFILE.TXT.

An alternate way of specifying the same file is the sequence

```
     MOVX AC1,GJ%OLD+GJ%SHT
     HRROI AC2,[ASCIZ/MYFILE.TXT/]
     GTJFN%
```

3.3.1.2 Long Form of GTJFN% - The long form of the GTJFN% monitor call requires the following two words of arguments:

```
         0                       17 18                     35
        !=====================================================!
  AC1   !             0            ! address of argument table !
        !=====================================================!
```

```
        0                                                      35
        !=======================================================!
   AC2  !   pointer to ASCIZ file specification string, or 0    !
        !=======================================================!
```

The argument block for the long form is described in Table 3-4.

Table 3-4: Long Form GTJFN% Argument Block

Word	Symbol	Meaning
0	.GJGEN	Flag bits appear in the left half and generation number appears in the right half.
1	.GJSRC	An input JFN appears in the left half and an output JFN appears in the right half. To omit either JFN, the user's program must specify the symbol .NULIO (377777).
2	.GJDEV	Pointer to ASCIZ string that specifies the device to be used when none is given. If this word is 0, DSK will be used.
3	.GJDIR	Pointer to ASCIZ string that specifies the directory to be used when none is given. If this word is 0, the user's connected directory will be used.
4	.GJNAM	Pointer to ASCIZ string that specifies the filename to be used when none is given. If this word is 0, the input must specify the filename.
5	.GJEXT	Pointer to ASCIZ string that specifies the file type to be used when none is given. If this word is 0, a null type will be used.
6	.GJPRO	Pointer to ASCIZ string or 3B2+octal protection code. This word indicates the protection to be used when none is given. If this word is 0, the protection as specified in the directory will be used.
7	.GJACT	Pointer to ASCIZ string or 3B2+decimal account number. This word indicates the account to be used when none is given. If this word is 0, the account specified when the user logged in will be used.
10	.GJJFN	The JFN to assign to the file specification if flag bit GJ%JFN is set in word .GJGEN (word 0) of the argument block.
11-17		Additional words allowed if flag bit GJ%XTN (bit 15) is set in word .GJGEN (word 0) of the argument block. These additional words are used when performing command input parsing and are described in the TOPS-20 Monitor Calls Reference Manual.

The flag bits accepted in the left half of .GJGEN (word 0) of the argument block are the same as those accepted in the short form of the GTJFN% call. The entire set of flag bits is listed in Table 3-2.

The generation number values accepted in the right half of .GJGEN (word 0) of the argument block can be 0, -1, -2, -3, or a specified number, although 0 is the normal case. Refer to Bits 18-35 of Table 3-2 for explanations of these values.

If execution of the GTJFN% call is successful, the JFN assigned is returned in the right half of AC1 and various bits are set in the left half if flag bits 11, 12 or 13 were on in the call. Refer to Table 3-3 for the explanations of the bits returned. Execution of the program continues at the second instruction following the call.

If execution of the GTJFN call is not successful, the JFN is not assigned and an error code is returned in the right half of AC1. The execution of the program continues at the instruction following the GTJFN% call.

The following sequence of instructions obtains a specification for an existing file from the user's terminal, assigns the JFN to the next higher generation of that file, and specifies default fields to be used if the user omits a field when he gives his file specification.

```
        MOVEI AC1,JFNTAB
        SETZ AC2,
        GTJFN%
         .
         .
         .
```

```
JFNTAB:  GJ%FOU
         XWD .PRIIN,.PRIOU
         0
         POINT 7,[ASCIZ/TRAIN/]   ;default directory
         0
         POINT 7,[ASCIZ/MEM/]     ;default file type
         0
         0
         0
```

The address of the argument table for the GTJFN% call (JFNTAB) is given in the right half of AC1. AC2 contains 0, which means no pointer to a string is given; thus, fields for the file specification will be taken only from the user's terminal. The first word of the argument block contains a flag bit for the GTJFN% call. This bit (GJ%FOU) indicates that the next higher generation number is to be assigned to the file. The second word of the argument block indicates that the file specification is to be obtained from the user's terminal, and any output generated because of the user employing recognition is to be printed on his terminal. If the user does not supply a directory name as part of his file specification, the directory <TRAIN> will be used. And if the user does not give a file type, the type MEM will be used. If the user omits other fields from his specification, the system standard value (refer to Table 3-1) will be used.

3.3.1.3 Summary of GTJFN% - The GTJFN% monitor call is required to associate a JFN with a particular file. In most cases, the short form of the GTJFN% call is sufficient for establishing this association. However, the long form is more powerful because it provides the user's program more control over the file specification that is obtained. The following summary compares the characteristics of the two forms of the GTJFN% monitor call.

Short Form	Long Form
Assigns a JFN to a file. System decides the JFN to assign.	Assigns a JFN to a file. User program may request a particular JFN.
Accepts the file specification from a string in memory or a file.	Accepts the file specification from a string in memory and a file.
Uses standard system values for fields not given in the file specification.	Allows user-supplied values to be used for fields not given in the file specification.

3.4 OPENING A FILE

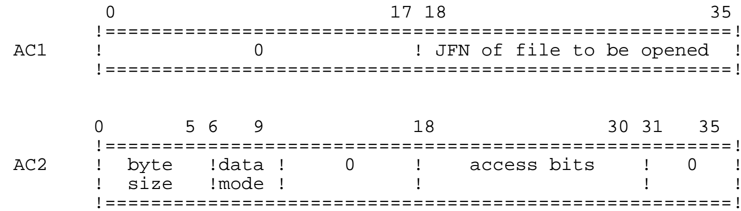
Once a JFN has been obtained for a file, the user's program must open the file in order to transfer data. The user's program supplies the JFN of the file to be opened and a word of bits indicating the desired byte size, data mode, and access to the file.

The desired access to the file is specified by a separate bit for each type of access. The file is successfully opened only if the desired access does not conflict with the current access to the file (refer to Section 3.1). For example, if the user requests both read and write access to the file, but write access is not allowed, then the file is not opened for this user. The allowed types of access to a file are:

- o Read access. The file can be read with byte, string, or random input.
- o Write access. The file can be written with byte, string, or random output.
- o Append access. The file can be written only with sequential byte or dump output, and the current byte pointer (refer to Section 3.5.1) cannot be changed. The initial position of the file pointer is at the end of the file.
- o Frozen access. The file can be concurrently accessed by at most one user writing the file, but by any number of users reading the file. This is the default access to a file.
- o Thawed access. The file can be accessed even if other users are reading and writing the file.
- o Restricted access. The file cannot be accessed if another user already has opened the file.
- o Unrestricted read access. The file can be read regardless of what other users might be doing with the file.

3.4.1 OPENF% Monitor Call

The OPENF% (Open File) monitor call opens a specified file. It requires the following two words of arguments.



If the left half of AC1 is not 0, the contents of AC1 is interpreted as a pointer to a string, not as a JFN. If the user's program requests bits returned in AC1 from the GTJFN% call, these bits must be cleared before executing the OPENF% call.

The byte size (OF%BSZ) in AC2 specifies the number of bits in each byte of the file and can be between 1 and 36 (decimal). If this field is 0 a byte size of 36 (decimal) is assumed.

The file data mode field (OF%MOD) usually has one of two values:

Value	Meaning
0	Normal data mode of the file (that is, byte I/O). Dump I/O is illegal.
17	Dump mode (that is, unbuffered word I/O). Byte I/O is illegal and the byte size is ignored.

The access bits are described in Table 3-5.

Table 3-5: OPENF% Access Bits

Bit	Symbol	Meaning
0-5	OF%BSZ	Byte size (maximum of 36 decimal).
6-9	OF%MOD	Data mode in which to open file.
18	OF%HER	Halt on the occurrence of an I/O device or medium error during subsequent I/O to the file. If this bit is not set, a software interrupt is generated if a device or medium error occurs during subsequent I/O.
19	OF%RD	Allow read access.
20	OF%WR	Allow write access.
21	OF%EX	Allow execute access.
22	OF%APP	Allow append access.
23	OF%RDU	Allow unrestricted read access.
24		Reserved for Digital.
25	OF%THW	Allow thawed access. If this bit is not set, the file is opened for frozen access.
26	OF%AWT	Block (that is, temporarily suspend) the program until access to the file is permitted.
27	OF%PDT	Do not update the access dates of the file.
28	OF%NWT	Return an error if access to the file cannot be permitted.
29	OF%RTD	Allow access to the file to only one process (that is, restricted access).
30	OF%PLN	Do not check for line numbers in the file.
31	OF%DUD	Suppress system updating of modified pages in memory to thawed files on disk unless CLOSF or UFPGS issued.
32	OF%OFL	Open device even if off-line.
33	OF%FDT	Force update of .FBREF (last read) in FDB and increment RH of .FBCNT (number of references).
34	OF%RAR	Wait if file off-line.

If bits OF%AWT and OF%NWT are both off, an error code is returned if access to the file cannot be permitted (that is, the action taken is identical to OF%NWT being on).

If execution of the OPENF% monitor call is successful, the file is opened, and the execution of the program continues at the second instruction after the OPENF% call.

If execution of the OPENF% call is not successful, the file is not opened, and an error code is returned in AC1. The execution of the program continues at the next instruction after the OPENF% call.

Two samples of the OPENF% call follow.

The sequence of instructions below opens a file for input.

```
HRRZ AC1,JFNEXT
MOVX AC2,FLD(44,OF%BSZ)+OF%RD+OF%PLN
OPENF%
```

The JFN of the file to be opened is contained in the location indicated by the address in AC1 (JFNEXT). The bits specified for AC2 indicate that the byte size is one word FLD(44,OF%BSZ), that read access is being requested to the file (OP%RD), and that no check will be made for line numbers in the file; that is, the line numbers will not be discarded (OF%PLN). Because bit OF%THW is not set, the file can be accessed for reading by any number of processes.

The following sequence of instructions can be used to open a file for output.

```
MOVE AC1,JFN
MOVX FLD(7,OF%BSZ)+OF%HER+OF%WR+OF%AWT
OPENF%
```

The right half of AC1 contains the address that has the JFN of the file to be opened. The bits specified for AC2 indicate that the byte size is 7-bit bytes FLD(7,OF%BSZ), that the program is to be halted when an I/O error occurs in the file (OF%HER), that write access is being requested to the file (OF%WR), and that the program is to be blocked if access cannot be granted (OF%AWT). Because bit OF%THW is not set, if another user has been granted write access to the file, this user's program will be blocked until access can be granted.

3.5 TRANSFERRING DATA

Data transfers of sequential bytes are the most common form of transfer and can be used with any file. For disk files, nonsequential bytes and entire pages can also be transferred.

3.5.1 File Pointer

Every open file is associated with a pointer that indicates the last byte read from or written to the file. When the file is initially opened, this pointer is normally positioned before the beginning of the file so that the first data operation will reference the first byte in the file. The pointer is then advanced through the file as data is transferred. However, if the file is opened for append-only access (bit OF%APP set in the OPENF% call), the pointer is positioned after the last byte of the file. This allows the first write operation to append data to the end of the file.

For disk files, the pointer may be repositioned arbitrarily throughout the file, such as in the case of nonsequential data transfers. When the pointer is positioned beyond the end of the file, an end-of-file indication is returned when the program attempts a read operation using byte input. When the program performs a write operation beyond the end of the file using byte output, the end-of-file indicator is updated to point to the end of the new data. However, if the program writes pages beyond the end of the file with the PMAP% monitor call (refer to section 3.5.6), the byte count is not updated. Therefore, it is possible for a file to contain pages of data beyond the end-of-file indicator. To allow sequential I/O to be performed later to the file, the program should update the byte count before closing the file. (Refer to the CHFDB% monitor call description in the _TOPS-20 Monitor Calls Reference Manual_.)

3.5.2 Source and Destination Designators

Because I/O operations occur by moving data from one location to another, the user's program must supply a source and a destination for any I/O operation. The most commonly-used source and destination designators are the following:

1. A JFN associated with a particular file. The JFN must be previously obtained with the GTJFN% or GNJFN% monitor call before it can be used.

2. The primary input and output designators .PRIIN and .PRIOU, respectively (refer to Section 2.2). These designators should be used when referring to the terminal.

3. A byte pointer to the beginning of the string of bytes in the program's address space that is being read or written. The byte pointer can take one of two forms:

   - A word with a -1 in the left half and an address in the right half. This form is used to designate a 7-bit ASCIZ string starting in the left-most byte of the specified address. A word in this form is functionally equivalent to a word assembled by the POINT 7,ADR pseudo-op.

- A full word byte pointer with a byte size of 7 bits.

Most monitor calls dealing with strings deal specifically with ASCII strings. Normally, ASCII strings are assumed to terminate with a byte of 0 (that is, are assumed to be ASCIZ strings). However some calls optionally accept an explicit byte count and/or terminating byte. These calls are generally ones that handle non-ASCII strings and byte sizes other than 7 bits.

3.5.3 Transferring Sequential Bytes

The BIN% (Byte Input) and BOUT% (Byte Output) monitor calls are used for sequential byte transfers. The BIN% call takes the next byte from the given source and places it in AC2. The BOUT% call takes the byte from AC2 and writes it to the given destination. The size of the byte is that given in the OPENF% call for the file.

The BIN% monitor call accepts a source designator in AC1, and upon successful execution of the call, the byte is right-justified in AC2. If execution of the call is not successful, an illegal instruction trap is generated. Control returns to the user's program at the instruction following the BIN% call. If the end of the file is reached, AC2 contains 0 instead of a byte. The program can process this end-of-file condition if a jump style error return is the next instruction following the BIN% call.

The BOUT% monitor call accepts a destination designator in AC1 and the byte to be output, right-justified in AC2. Upon successful execution of the call, the byte is written to the destination. If execution of the call is not successful, an illegal instruction trap is generated Control returns to the user's program at the instruction following the BOUT% call.

The following sequence shows the transferring of bytes from an input file to an output file. The bytes are read from the file indicated by INJFN and written to the file indicated by OUTJFN.

```
LOOP:  MOVE 1,INJFN       ;get source designator from INJFN
       BIN%               ;read a byte from the source
       ERJMP DONE         ;check for end of file, if 0
LOOP2: MOVE 1,OUTJFN      ;get destination from OUTJFN
       BOUT%              ;write the byte to the destination
       JRST LOOP          ;continue until 0 byte is found
DONE:  GTSTS%             ;obtain status of source
       TXNN 2,GS%EOF      ;test for end of file
       JRST NOTYET        ;no, test for 0 in input file
        :                 ;yes, process end of file condition
NOTYET:MOVEI 2,0          ;0 in input file
       JRST LOOP2
```


3.5.4 Transferring Strings

The SIN% (String Input) and SOUT% (String Output) monitor calls are used for string transfers. These calls transfer either a string of a specified number of bytes or a string terminated with a specific byte.

The SIN% monitor call reads a string from the specified source into the program's address space. The call accepts four words of arguments in AC1 through AC4.

AC1: source designator

AC2: pointer to area in program's address space

AC3: count of number of bytes to read, or 0

AC4: byte on which to terminate input (optional)

The contents of AC3 are interpreted as the number of characters to read.

- If AC3 is 0, then reading continues until a 0 byte is found in the input.

- If AC3 is positive, then reading continues until either the specified number of bytes is read, or a byte equal to that given in AC4 is found in the input, whichever occurs first.

- If AC3 is negative, then reading continues until minus the specified number of bytes is read.

The contents of AC4 needs to be specified only if the contents of AC3 is a positive number. The byte in AC4 is right-justified.

The input is terminated when one of the following occurs:

- The byte count becomes zero.

- The specified terminating byte is reached.

- The end of the file is reached.

- An error occurs during the transfer (for example, a data error occurs).

Control returns to the user's program at the instruction following the SIN% call. If an error occurs (including the end of the file is reached), an illegal instruction trap is generated. In addition, several locations are updated:

1. The position of the file's pointer is updated for subsequent I/O to the file.

2. The pointer to the string in AC2 is updated to reflect the last byte read or, if AC3 contained 0, the last nonzero byte read.

3. The count in AC3 is updated, if pertinent, by subtracting the number of bytes actually read from the number of bytes requested to be read (that is, the count is updated toward zero). From this count, the user's program can determine the number of bytes actually transferred.

The SOUT% monitor call writes a string from the program's address space to the specified destination. Like the SIN% call, this call accepts four words of arguments in AC1 through AC4.

AC1: destination designator

AC2: pointer to string to be written

AC3: count of the number of bytes to write, or 0

AC4: byte on which to terminate output (optional)

The contents of AC3 and AC4 are interpreted in the same manner as they are in the SIN% monitor call.

The transfer is terminated when one of the following occurs.

- The byte count becomes zero.

- The specified terminating byte is reached. This terminating byte is written to the destination.

- An error occurs during the transfer.

Control returns to the user's program at the instruction following the SOUT% call. If an error occurs, an illegal instruction trap is generated. In addition, the position of the file's pointer, the pointer to the string in AC2, and the count in AC3, if pertinent, are also updated in the same manner as in the SIN% monitor call.

The following code sequence shows transferring a string from an input file to an output file. The procedure is the same as at the end of Section 3.5.3, using SIN% and SOUT% calls instead of BIN% and BOUT%.

```
LOOP:  MOVE 1,INJFN       ;get source from INJFN
       HRROI 2,BUF128     ;pointer to string to read into (128
                          ;word buffer)
```



```
       MOVNI 3,^D128*5    ;input a maximum of 640 bytes
       SIN%               ;transfer until end of buffer or end of
                          ;file
        ERCAL EOFQ        ;error occurred

       ADDI 3,^D128*5     ;determine negative number of
                          ;bytes transferred
       MOVN 3,3           ;convert to positive
       MOVE 1,OUTJFN      ;get destination from OUTJFN
       HRROI 2,BUF128     ;pointer to string to write from
       SOUT%              ;transfer as many bytes as read
EOFQ:  MOVE 1,INJFN
       GTSTS%             ;obtain status of source
       TXNN 2,GS%EOF      ;test for end of file
       RET                ;no, continue copying
```

3.5.5 Transferring Nonsequential Bytes

As discussed in Section 3.5.3, the BIN% and BOUT% calls transfer bytes sequentially, starting at the current position of the file's pointer. The RIN% (Random Input) and ROUT% (Random Output) monitor calls allow the user's program to specify where the transfer will begin by accepting a byte number within the file. The size of the byte is the size given in the OPENF% call for the file. The RIN% and ROUT% calls can only be used when transferring data to or from disk files.

The RIN% monitor call takes a byte from the specified location in the file and places it into the accumulator. The call accepts the JFN of the file in AC1 and the byte number within the file in AC3. Upon successful completion of the call, the byte is right-justified in AC2, and the file's pointer is updated to point to the byte following the one just read. If an error occurs, an illegal instruction trap is generated. Control returns to the user's program at the instruction following the RIN% call.

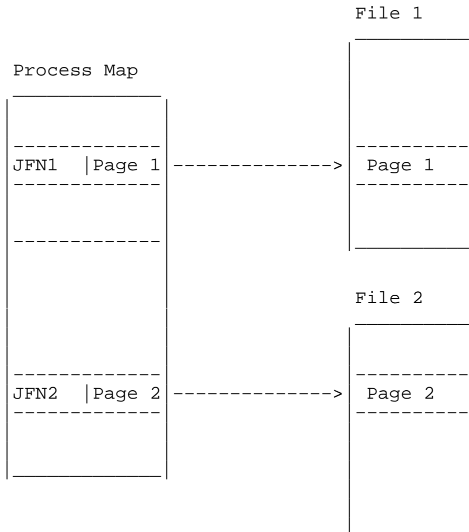
The ROUT% monitor call takes a byte from the accumulator and writes it into the specified location in the file. The call accepts the JFN of the file in AC1, the byte to write right-justified in AC2, and the byte number within the file in AC3. Upon successful completion of the call, the byte is written into the specified byte in the file, and the file's pointer is updated to point to the byte following the one just written. If an error occurs, an illegal instruction trap is generated. Control returns to the user's program at the instruction following the ROUT% call.

3.5.6 Mapping Pages

Up to this point, monitor calls have been presented for transferring

bytes and strings of data. The next call to be discussed is used to transfer entire pages of data between a file and a process.

Both files and process address spaces are divided into pages of 512(decimal) words. A page within a file can be identified by one word, where the JFN of the file is in the left half and the page number within the file is in the right half. A page within a process address space can also be identified by one word, where the identifier of the process (refer to Section 5.3) is in the left half and the page number within the process' address space is in the right half. Each one-word identifier for the pages in the process address space is placed in what is called the process page map. When identifiers for file pages are placed in the process page map, references to the process page actually refer to the file page. The following diagram illustrates a process map that has identifiers for pages from two files.



The PMAP% (Page Mapping) monitor call is used to map one or more entire pages from a file to a process (for input), from a process to a file (for output), or from one process to another process. In general, this call changes the entries in the process map by accepting

file page identifiers and process page identifiers as arguments. Mapping pages between a file and a process is described below; mapping pages between two processes is described in Chapter 5.

3.5.6.1 Mapping File Pages to a Process - This use of the PMAP% call changes the map of the process so that references to pages in the process reference pages in a file. This does not actually cause data to be transferred; it simply changes the contents of the map. Later when changes are made to the actual page in the process, the changes will also be made to the page in the file, if write access has been specified for the file.

Note that you cannot map file pages to pages in a process section that does not exist in the the process map. If you use PMAP% to input file pages to pages in a nonexistent section of a process, the monitor generates an illegal instruction trap.

In addition, you can map one or more file sections (of 512 pages each) into a process. See Section 8.3.1 for details.

The PMAP% call accepts three words of arguments in AC1 through AC3.

AC1: JFN of the file in the left half, and the page number in the file in the right half

AC2: process identifier (refer to Section 5.3) in the left half, and page number in the process in the right half

AC3: repetition count and access

The repetition count and access bits that can be specified in AC3 are described in Table 3-6.

Table 3-6: PMAP% Access Bits

Bit	Symbol	Meaning
0	PM%CNT	Repeat the mapping operation the number of times specified by the right half of AC3. The file page number and the process page number are incremented by 1 each time the operation is performed.
2	PM%RD	Allow read access to the page.
3	PM%WR	Allow write access to the page.

4	PM%EX	Reserved. The symbol PM%RWX can be used to set B2-4.
5	PM%PLD	Preload page being mapped (move the page immediately instead of waiting until it is referenced).
9	PM%CPY	Create a private copy of the page if the process writes into the page. This is called copy-on-write and causes the map to be changed so that it identifies the copy instead of the original. Write access is allowed to the copy even if it was not allowed to the original. This allows a process to change a page of data without changing the data for other processes that have also mapped the page.
10	PM%EPN	Bits 18-35 of AC2 contain extended (18-bit) process page number. If the section containing the page does not exist, a private section is created.
11	PM%ABT	Unmap page and discard (abort) changed contents.
18-35	PM%RPT	The number of times to repeat the mapping operation if bit 0(PM%CNT) is set.

With this use of the PMAP% call, the present contents of the page in the process are removed. If the page in the file is currently nonexistent, it will be created when it is written.

This use of the PMAP% call is valid only if the file is opened for at least read access. If write access is requested in the PMAP% call, it is not granted unless it was also specified in the OPENF% call when the file was opened.

A file cannot be closed while any of its pages are mapped into any process. Thus, before a file is closed, its pages must be unmapped (refer to Section 3.5.6.3).

After execution of the PMAP% call, control returns to the user's program at the instruction following the call. If an error occurs, an illegal instruction trap is generated.

3.5.6.2 Mapping Process Pages to a File - This use of the PMAP% call actually transfers data by moving the specified page in the process to the specified page in the file. The process map for the page is now

empty. Both the page in the process and the page in the file must be private; that is, no other process can have the page mapped into its address space. The ownership of the process page is transferred to the file page. The previous contents of the page in the file are deleted.

The three words of arguments are as follows:

AC1: process identifier (refer to Section 5.3) in the left half, and page number in the process in the right half

AC2: JFN of the file in the left half, and the page number in the file in the right half

AC3: repetition count and access (refer to Section 3.5.6.1)

The access requested in the PMAP% call is granted only if it does not conflict with the access specified in the OPENF% call when the file was opened.

This use of the PMAP% call does not automatically update the files byte count and the byte size. To allow the file to be read later with sequential I/O monitor calls, the program should update the file's byte count and the byte size. (Refer to the CHFDB% monitor call in the TOPS-20 Monitor Calls Reference Manual).

3.5.6.3 Unmapping Pages in a Process - As stated previously, a file cannot be closed if any of its pages are mapped in any process. To unmap a file's pages from a process, the program must execute the SMAP% call, or the following form of the PMAP% call:

AC1: -1

AC2: process identifier in the left half, and page number in the process in the right half.

AC3: the repeat count for the number of pages to remove from the process (refer to Section 3.5.6.1).

3.5.7 Mapping File Sections to a Process

A section of memory is a unit of 512 pages of process address space. File sections also contain 512 pages. The first page of each file section has a page number that is an integral multiple of 512. Like memory pages, sections can be mapped from one process to another, from a process to itself, or from a file to a process. Chapter 8 describes the SMAP% call completely.

The SMAP% (Section Mapping) monitor call is similar to the PMAP% call. The SMAP% call maps one or more sections from a file to a process (for input), or from one process to another process. To map a process section to a file, you must use the PMAP% call as described in Chapter 5 to map each page.

Mapping a file section to a process section with SMAP% does not cause data to move from the disk to memory. Instead, SMAP% changes the contents of the process memory map so that the process section pointer points to a file section. The monitor transfers data only when your program references a memory page to which a file page is mapped.

To map a file section to a process section, SMAP% requires three arguments:

AC1: source identifier: a JFN in the left half, and a file section number in the right half. If several contiguous sections are to be mapped, the number in the right half is that of the first section in the group of contiguous sections.

AC2: destination identifier: process identifier in the left half, and a process section number in the right half. If several contiguous sections are to be mapped, the number in the right half is the number of the first section into which SMAP% maps a file section.

AC3: flags that control access to the process section in the left half, and, in the right half, the number of sections to map into the process. The number of sections to map cannot be less than 1 nor more than 32 (decimal).

The flags in the left half of AC3 are described in Table 3-7.

Table 3-7: SMAP% Access Bits

Bit	Symbol	Meaning
2	SM%RD	Allow read access.
3	SM%WR	Allow write access.
4	SM%EX	Allow execute access.
6	SM%IND	Map the destination section using an indirect section pointer.

3.6 CLOSING A FILE

Once data has been transferred to or from a file, the user's program must close the file. When a file is closed, the system automatically performs the following:

1. Updates the directory information for the file. For example, for a file to which sequential bytes had been written, the byte size and byte count are updated when the file is closed.

2. Releases the JFN associated with the file. However, the user's program can request to close the file, but retain the JFN assignment. This is useful if the program plans to reopen the same file later, but does not want to execute another GTJFN% call.

3.6.1 CLOSF% Monitor Call

The CLOSF% (Close File) monitor call closes either the specified file or all files that are opened for the process executing the call. The CLOSF% call accepts one word of arguments in AC1 - flag bits in the left half and the JFN of the file to be closed in the right half. The flag bits are described in Table 3-8.

Table 3-8: CLOSF% Flag Bits

Bit	Symbol	Meaning
0	CO%NRJ	Do not release the JFN from the file.
6	CZ%ABT	Abort any output operations currently being done. That is, close the file but do not perform normal cleanup operations (for example, do not output any data remaining in the buffers). If output to a new disk file that has not been closed is aborted, the file is closed and then deleted.
7	CS%NUD	Do not update the copy of the directory on the disk (refer to the CHFDB% description in the _TOPS-20 Monitor Calls Reference Manual_ for more information).

If the contents of AC1 is -1, all files that are opened for this process are closed.

If the execution of the CLOSF% call is successful, the specified file is closed, and the JFN associated with the file is released if CO%NRJ was not set in the call. The execution of the user's program continues at the second location after the CLOSF% call.

If the execution of the CLOSF% call is not successful, the file is not closed and an error code is returned in the right half of AC1. The execution of the user's program continues at the instruction following the CLOSF% call.

The following sequence illustrates the closing of two files.

```
CLOSIF:  HRRZ 1,INJFN    ;obtain input JFN
         CLOSF%          ;close input file
          ERJMP FATAL    ;if error, print message and stop
CLOSOF:  HRRZ 1,OUTJFN   ;obtain output JFN
         CLOSF%          ;close output file
          ERJMP FATAL    ;if error, print message and stop
```

3.7 ADDITIONAL FILE I/O MONITOR CALLS

3.7.1 GTSTS% Monitor Call

The GTSTS% (Get Status) monitor call obtains the status of a file. This call accepts one argument word - the JFN of the file in the right half of the AC1. The left half of AC1 is zero.

Control always returns to the user's program at the instruction following the GTSTS% call. Upon return, appropriate bits reflecting the status of the specified JFN are set in AC2. These bits, and their meanings, are described in Table 3-9. Note that if the JFN is illegal or unassigned, bit 10 (GS%NAM) will not be set.

Table 3-9: Bits Returned on GTSTS% Call

Bit	Symbol	Meaning
0	GS%OPN	The file is open. If this bit is not set, the file is not open.
1	GS%RDF	If the file is open (for example, GS%OPN is set), it is open for read access.
2	GS%WRF	If the file is open, it is open for write access.
3	GS%XCF	File is open for execute access.
4	GS%RND	If the file is open, it is open for non-append access (that is, its pointer can be reset).
5-6		Reserved for Digital.
7	GS%LNG	File has pages in existence beyond page number 511.
8	GS%EOF	The last read operation to the file was at the end of the file.
9	GS%ERR	The file may be in error (for example, the bytes read may be erroneous).
10	GS%NAM	A file specification is associated with this JFN. This bit will not be set if the JFN is in any way illegal.
11	GS%AST	One or more fields of the file specification associated with this JFN contain a wildcard character.
12	GS%ASG	The JFN is currently being assigned (that is, a process other than the one executing the GTSTS call is assigning this JFN).
13	GS%HLT	An I/O error is considered to be a terminating condition for this JFN. That is, the OPENF% call for this JFN had bit OF%HER set.
14-16		Reserved for Digital.
17	GS%FRK	Access to the file is restricted to only one process.
18	GS%PLN	If on, file line numbers are passed during input; if zero, line numbers are stripped before input.
19-31		Reserved for Digital.
32-35	GS%MOD	The data mode of the file (refer to the OPENF% call).

Value	Symbol	Meaning
0	.GSNRM	Normal (sequential) I/O
1	.GSSMB	Small buffer mode
10	.GSIMG	Image (binary) I/O
17	.GSDMP	Dump I/O

An example of the GTSTS% call is shown in the first program in Section 3.9.

3.7.2 JFNS% Monitor Call

The JFNS% (JFN to String) monitor call returns the file specification currently associated with the specified JFN. The call accepts three words of arguments in AC1 through AC3.

AC1: destination designator where the file specification associated with the JFN is to be written. This specification is an ASCIZ string.

AC2: JFN or pointer to string (see below)

AC3: format to be used when returning the specification (see below)

The contents of AC1 can be any valid destination designator (refer to Section 3.5.2).

The contents of AC2 can be one of two formats. The first format is a word with either flag bits or 0 in the left half and the JFN in the right half. The bits that can be given in the left half of AC2 are the ones returned from the GTJFN% call (refer to Table 3-3). When the left half of AC2 is nonzero (that is, contains the bits returned from the GTJFN% call), the string returned will contain wildcard characters for appropriate fields and 0, -1, or -2 as a generation number if the corresponding bit is on in the JFNS% call. When the left half of AC2 is 0, the string returned is the exact specification for the file (for example, wildcard characters are not returned for any fields). If the JFN is associated only with a file specification and not with an actual file (that is, bit GJ%OFG was set in the GTJFN% call), the string returned will contain null fields for unspecified fields and the actual values for specified fields. The second format allowed for AC2 is a pointer to the string in the program's address space that is to be returned upon execution of the call. Refer to the TOPS-20 Monitor Calls Reference Manual for the explanation of this format.

The contents of AC3 specify the format in which the specification is written to the destination. Bits 0 through 20 are divided into 3-bit bytes, each byte representing a field in the file specification. The value of the byte indicates the format for that field. The possible values are:

Value	Symbol	Meaning
0	.JSNOF	Do not return this field when returning the file specification.
1	.JSAOF	Always return this field when returning the file specification.
2	.JSSSD	Suppress this field if it is the standard system value for this field (refer to Table 3-1).

If the contents of AC3 is zero, the file specification is written in the format

dev:<directory>name.typ.gen;T

with fields the same as the standard system value (see Table 3-1) not returned and protection and account fields returned only if bit 9 and bit 10 in AC2 are on, respectively. The temporary attribute (;T) is returned only if the file is temporary.

Table 3-10 describes the bits that can be set in AC3.

Table 3-10: JFNS% Format Options

Bit	Symbol	Meaning
0	JS%NOD	Print node name if node name is present.
1-2	JS%DEV	Format for device field.
3-5	JS%DIR	Format for directory field.
6-8	JS%NAM	Format for filename field. A value of 2 (that is, bit 7 set) for this field is illegal.
9-11	JS%TYP	Format for file type field. A value of 2 (that is, bit 10 set) for this field is illegal.

12-14	JS%GEN	Format for generation number field.
0-14	JS%SPC	Output for all file specification fields named above. This field should have the same bits set as would be set in the fields above. (See B35 (JS%PAF) below.)
15-17	JS%PRO	Format for protection field.
18-20	JS%ACT	Format for account field.
21	JS%TMP	Return temporary file indication ;T if the file specification is for a temporary file.
22	JS%SIZ	Return size of file in pages (see below).
23	JS%CDR	Return creation date of file (see below).
24	JS%LWR	Return date of last write operation to file (see below).
25	JS%LRD	Return date of last read operation from file (see below).
26	JS%PTR	AC2 contains a pointer to the string containing the field to be returned (refer to the _TOPS-20 Monitor Calls Reference Manual_ for a description of this use of the JFNS% call).
27	JS%ATR	Return file specification attributes if appropriate.
28	JS%AT1	Return specification attribute referenced in AC4.
29	JS%OFL	Return the "OFF-LINE" attribute.
30-31		Reserved for Digital.
32	JS%PSD	Punctuate the size and date fields (see below) in the file specification returned.
33	JS%TBR	Place a tab before all fields returned (that is, fields whose value is given as 1 in the 3-bit field) in the file specification, except for the first field.

34	JS%TBP	Place a tab before all fields that may be returned (that is, fields whose value is given as 1 or 2 in the 3-bit field) in the file specification, except for the first field.
35	JS%PAF	Punctuate all fields (see below) returned in the file specification from the device field through the ;T field. If bits 32 through 35 are not set, no punctuation is used between the fields.

The punctuation used on each field is shown below.

dev:<directory>name.typ.gen;A(account);P(protection);T(temporary)
,size,creation date,write date,read date

Refer to Section 1.2.2 for information on error returns.

3.7.3 GNJFN% Monitor Call

Occasionally a program may be written to perform similar operations on a group of files instead of only on one file. However, the program should not require the user to give a file specification for each file. Because the GTJFN% call associates a JFN with only one file at a time, the program needs a method of assigning a JFN to all the files in the group. By using the GTJFN% call to initially obtain the JFN and the GNJFN% call to assign the same JFN to each subsequent file in the group, a program can accept a specification for a group of files and process each file in the group individually. After the user gives the initial file specification, the program requires no additional input.

Before an example showing the interaction of these two calls is given, a description of the GNJFN% (Get Next JFN) monitor call is appropriate.

The GNJFN% monitor call assigns a JFN to the next file in a group of files that have been specified with wildcard characters. The next file is determined by searching the directory in the order described in Section 3.3.1.1 using the current file as the first item. This call accepts one argument word in AC1 - the flags returned from the GTJFN% call in the left half and the JFN of the current file in the right half. In other words, the information returned in AC1 from the GTJFN% call is given as an argument to the GNJFN% call. Therefore, the program must save this information for use with the GNJFN% call.

If execution of the GNJFN% call is successful, the same JFN is assigned to the next file in the group. The left half of AC1 contains various flags and the right half contains the JFN. The execution of the program continues at the second instruction after the GNJFN% call.

Table 3-11 describes the bits that can be returned in AC1 on a successful GNJFN% call.

Table 3-11: GNJFN% Return Bits

Bit	Symbol	Meaning
13	GN%STR	A change in structure occurred between the previous file and this file.
14	GN%DIR	A change in directory occurred between the previous file and this file.
15	GN%NAM	A change in filename occurred between the previous file and this file.
16	GN%EXT	A change in file type occurred between the previous file and this file. If GN%NAM is on, this bit will also be on because the system considers two files with different filenames but with the same file type as a change in both the name and type.

If execution of the GNJFN% call is not successful, an error code is returned in the right half of AC1. Conditions that can cause an error return are:

1. The file currently associated with the JFN must be closed, and it is not. This means that the program must execute a CLOSF% call (with CO%NRJ set to retain the JFN) before executing a GNJFN% call.

2. There are no more files in this group. This return occurs on the first GNJFN% call after all files in the group have been stepped through. The JFN is released when there are no more files. (Note: This error may occur if the file currently associated with the JFN is deleted or renamed.)

The execution of the program continues at the next instruction after the GNJFN% call.

Consider the following situation. The user wants to write a program that will accept from his terminal a specification for a group of files and then perform an operation on each file individually without requiring additional input. Assume the user's directory <TRAIN> contains the following files:

```
FIRST.MAC.1
FIRST.REL.1
SECOND.REL.1
THIRD.EXE.1
```

As discussed in Section 3.3.1.1, a group of files can be given to the GTJFN call by supplying a specification that contains wildcard characters in one or more of its fields. Thus, the specification

```
<TRAIN>*.*
```

would refer to all four files in the user's directory <TRAIN>.

In his program, the user includes a GTJFN% call that will accept the above specification.

The call is

```
MOVX AC1,GJ%OLD+GJ%IFG+GJ%FLG+GJ%FNS+GJ%SHT
MOVE AC2,[.PRIIN,,.PRIOU]
GTJFN%
```

and indicates that

1. The file specification given must refer to an existing file (GJ%OLD).

2. The file specification given is allowed to contain wildcard characters (GJ%IFG).

3. Flags will be returned in AC1 on a successful call (GJ%FLG). The flags must be returned because they will be given to the GNJFN% call as arguments.

4. The contents of AC2 will be interpreted as containing an input and output JFN (GJ%FNS).

5. The short form of the GTJFN% call is being used (GJ%SHT).

6. The file specification is to be read from the user's terminal (.PRIIN,,.PRIOU).

When the user types the specification <TRAIN>*.* as input, the system associates the JFN with one file only. This file is the first one found when searching the directory in the order specified in Section 3.3.1.1. Thus the JFN returned is associated with the file FIRST.MAC.1.

After the GTJFN% call is successfully executed, AC1 contains appropriate flags in the left half and the JFN assigned in the right half. The flags that will be returned in this particular situation are:

GJ%NAM (bit 3)	A wildcard character appeared in the name field of the file specification given.
GJ%EXT (bit 4)	A wildcard character appeared in the type field of the file specification given.
GJ%GND (bit 12)	Any files marked for deletion will not be considered.

These flags inform the program of the fields that contained wildcard characters. The user's program must now save the contents of AC1 because this word will be used as the argument to the GNJFN% call. The program then performs its desired operation on the first file. Once its processing is completed, the program is ready for the specification of the next file. But instead of requesting the specification from the user, the program executes the GNJFN% call to obtain it. The argument to the GNJFN% call is the contents of AC1 returned from the previous GTJFN% call. Thus, the call in this case is equivalent to:

```
MOVE AC1,[GJ%NAM+GJ%EXT+GJ%GND,,JFN]
GNJFN%
```

Upon successful execution of the GNJFN% call, the JFN is now associated with the next file in the group (that is, FIRST.REL.1). AC1 contains appropriate flags in the left half and the same JFN in the right half. In this example, the flag returned is GN%EXT (bit 16) to indicate that the file type changed between the two files.

After processing the second file, the user's program executes another GNJFN% call using the original contents of AC1 returned from the GTJFN% call. The original contents must be used because this word indicates the fields containing wildcard characters. If the current contents of AC1 (that is, the flags returned from the GNJFN% call) are used, a subsequent GNJFN% call would fail because there are no flags set indicating fields containing wildcard characters. This second GNJFN% call associates the JFN with the file SECOND.REL.1. The flags returned in AC1 are GN%NAM (bit 15) and GN%EXT (bit 16) indicating that the filename and file type changed between the two files. (Remember that a change in filename implies a change in file type even if the two file types are the same.)

After processing this third file, the user's program executes another GNJFN% call using the original contents of AC1. Upon execution of the call, the JFN is now associated with THIRD.EXE.1, and the flags returned are GN%NAM and GN%EXT, indicating a change in the filename and file type.

After processing the file THIRD.EXE.1, the user's program executes a final GNJFN% call. Since there are no more files in the group, the call returns an error code and releases the JFN. Execution of the user's program continues at the instruction following the GNJFN% call.

3.8 SUMMARY

To read from or write to a file, the user's program must:

1. Obtain a JFN on the file with the GTJFN% monitor call (refer to Section 3.3.1).

2. Open the file with the OPENF% monitor call (refer to Section 3.4.1).

3. Transfer the data with byte, string, or page I/O monitor calls (refer to Section 3.5).

4. Close the file with the CLOSF% monitor call (refer to Section 3.6.1).

3.9 FILE EXAMPLES

Example 1 - This program assigns JFNs, opens an input file and an output file, and copies data from the input file to the output file. Data is copied until the end of the input file is reached. Refer to the _TOPS-20 Monitor Calls Reference Manual_ for explanation of the ERSTR% monitor call.

```
;*** PROGRAM TO COPY INPUT FILE TO OUTPUT FILE. ***
;       (USING BIN%/BOUT% AND IGNORING NULLS)

        TITLE FILEIO            ;TITLE OF PROGRAM
        SEARCH MONSYM           ;SEARCH SYSTEM JSYS-SYMBOL LIBRARY
        SEARCH MACSYM
        .REQUIRE SYS:MACREL

;*** IMPURE DATA STORAGE AND DEFINITIONS ***


INJFN:  BLOCK 1                 ;STORAGE FOR INPUT JFN
OUTJFN: BLOCK 1                 ;STORAGE FOR OUTPUT JFN
```

```
        PDLEN=3                 ;STACK HAS LENGTH 3
PDLST:  BLOCK PDLEN             ;SET ASIDE STORAGE FOR STACK

        STDAC.                  ;DEFINE STANDARD ACs. SEE MACSYM.

;*** PROGRAM INITILIZATION ***

START:  RESET%                  ;CLOSE FILES, ETC.
        MOVE P,[IOWD PDLEN,PDLST] ;ESTABLISH STACK

;***  GET INPUT FILE ***

INFIL:                          ;PROMPT FOR INPUT FILE
        TMSG <
INPUT FILE: >                   ;ON CONTROLLING TERMINAL
        MOVX T1,GJ%OLD+GJ%FNS+GJ%SHT ;SEARCH MODES FOR GTJFN
                                ;EXISTING FILE ONLY, FILE-NRs IN B
                                ;SHORT CALL
        MOVE T2,[.PRIIN,,.PRIOU] ;GTJFN'S I/O WITH CONTROLLING TERM
        GTJFN%                  ;GET JOB FILE NUMBER (JFN)
         ERJMPS [ PUSHJ P,WARN  ;IF ERROR, GIVE WARNING
                  JRST INFIL]   ;AND LET HIM TRY AGAIN
        MOVEM T1,INJFN          ;SUCCESS, SAVE THE JFN

;*** GET OUTPUT FILE ***

OUTFIL:                         ;PRINT PROMPT FOR
        TMSG <
OUTPUT FILE: >                  ;OUTPUT FILE
        MOVX T1,GJ%FOU+GJ%MSG+GJ%CFM+GJ%FNS+GJ%SHT ;GTJFN SEARCH MODES
                                ;[DEFAULT TO NEW GENERATION, PRINT
                                ; MESSAGE, REQUIRE CONFIRMATION
                                ; FILE-NR'S IN T2, SHORT CALL ]
        MOVE T2,[.PRIIN,,.PRIOU] ;I/O WITH CONTROLLING TERMINAL
        GTJFN%                  ;GET JOB FILE NUMBER
         ERJMPS [ PUSHJ P,WARN  ;IF ERROR, GIVE WARNING
                 JRST OUTFIL]   ;AND LET HIM TRY AGAIN
        MOVEM T1,OUTJFN         ;SAVE THE JFN

;NOW, OPEN THE FILES WE JUST GOT

;   INPUT

        MOVE T1,INJFN           ;RETRIEVE THE INPUT JFN
        MOVX T2,FLD(7,OF%BSZ)+OF%RD ;MODES FOR OPENF
                                ;[7-BIT BYTES + INPUT]
        OPENF%                  ;OPEN THE FILE
         ERJMPS FATAL           ;IF ERROR, GIVE MESSAGE AND STOP

;   OUTPUT

        MOVE T1,OUTJFN          ;GET THE OUTPUT JFN
```



```
        MOVX T2,FLD(7,OF%BSZ)+OF%WR ;MODES FOR OPENF
                                ;[7-BIT BYTES + OUTPUT]
        OPENF%                  ;OPEN THE FILE
         ERJMPS FATAL           ;IF ERROR, GIVE MESSAGE AND STOP


;*** MAIN LOOP: COPY BYTES FROM INPUT TO OUTPUT ***

LOOP:   MOVE T1,INJFN           ;GET THE INPUT JFN
        BIN%                    ;TAKE A BYTE FROM THE SOURCE
        JUMPE T2,DONE           ;IF 0, CHECK FOR END OF FILE
        MOVE T1,OUTJFN          ;GET THE OUTPUT JFN
        BOUT                    ;OUTPUT THE BYTE TO DESTINATION
         ERCALS ERROR
        JRST LOOP               ;LOOP, STOP ONLY ON A 0 BYTE
                                ;(FOUND AT LOOP+2)


;*** TEST FOR END OF FILE, ON SUCCESS FINISH UP ***

DONE:   GTSTS%                  ;GET THE STATUS OF INPUT FILE
        TXNN T2,GS%EOF          ;AT END OF FILE?
        JRST LOOP               ;NO, FLUSH NULL AND CONTINUE COPY

CLOSIF: MOVE T1,INJFN           ;YES, RETRIEVE INPUT JFN
        CLOSF%                  ;CLOSE INPUT FILE
         ERJMPS FATAL           ;IF ERROR, GIVE MESSAGE AND STOP

CLOSOF: MOVE T1,OUTJFN          ;RETRIEVE OUTPUT JFN
        CLOSF%                  ;CLOSE OUTPUT FILE
         ERJMPS FATAL           ;IF ERROR, GIVE MESSAGE AND STOP
        TMSG <
[DONE]>                         ;SUCCESSFULLY DONE
        JRST ZAP                ;STOP

;*** ERROR HANDLING ***

FATAL:  TMSG <
?>                              ;FATAL ERRORS PRINT ? FIRST
        PUSHJ P,ERROR           ;THEN PRINT ERROR MESSAGE
        JRST ZAP                ;AND STOP

WARN:   TMSG <
%>                              ;WARNINGS PRINT % FIRST
                                ;AND FALL THRU 'ERROR'
                                ;BACK TO CALLER

ERROR:  MOVEI T1,.PRIOU         ;DECLARE PRINCIPAL OUTPUT DEVICE
                                ;FOR ERROR MESSAGE
        MOVE T2,[.FHSLF,,-1]    ;CURRENT FORK,, LAST ERROR
        SETZ T3,                ;NO LIMIT,, FULL MESSAGE
        ERSTR%                  ;PRINT THE MESSAGE
```

```
         JFCL                    ;IGNORE UNDEFINED ERROR NUMBER
         JFCL                    ;IGNORE ERROR DURING EXE OF ERSTR
        POPJ P,                  ;RETURN TO CALLER

ZAP:    HALTF%                   ;STOP
        JRST START               ;WE ARE RESTARTABLE
        END START                ;TELL LINKING LOADER START ADDRESS
```

Example 2 - This program accepts input from a user at the terminal and then outputs the data to the line printer. Refer to Section 2.9 for explanation of the RDTTY% call.

```
        TITLE LPTPNT             ;PROGRAM TO PRINT TERMINAL INPUT
                                 ;ON PRINTER
        SEARCH MONSYM            ;SEARCH SYSTEM JSYS-SYMBOL LIBRARY
        SEARCH MACSYM
        .REQUIRE SYS:MACREL

        STDAC.                   ;DEFINE STANDARD ACs

BUFSIZ==200
PDLEN==50

COUNT:  BLOCK 1
LPTJFN: BLOCK 1
BUFFER: BLOCK BUFSIZ
PDL:    BLOCK PDLEN

START:  RESET%                   ;RESET I/O, ETC.
        MOVE P,[IOWD PDLEN,PDL] ;SET UP STACK
        TMSG <ENTER TEXT TO BE PRINTED (END WITH ^Z):
>                                ;OUTPUT PROMPTING TEXT
        HRROI T1,BUFFER          ;GET POINTER TO BUFFER
        MOVE T2,[RD%BRK+BUFSIZ*5] ;GET FLAG AND MAX # OF CHARS TO READ
        SETZM T3                 ;NO RE-TYPE BUFFER
        RDTTY%                   ;INPUT TEXT FROM TERMINAL
         EJSHLT                  ;ERROR, STOP
        HRRZS T2                 ;GET CHARS REMAINING IN BUFFER
        MOVEI T1,BUFSIZ*5        ;COMPUTE NUMBER OF CHARS READ =
        SUB T1,T2                ;BUFFERSIZE MINUS CHARS REMAINING
        SOS T1                   ;DON'T INCLUDE ^Z
        MOVEM T1,COUNT           ;SAVE # OF CHARS INPUT
;GET A JFN FOR THE PRINTER AND OPEN THE PRINTER

        MOVX T1,GJ%SHT!GJ%FOU    ;OUTPUT FILE, SHORT CALL
        HRROI T2,[ASCIZ /LPT:/] ;GET POINTER TO NAME OF FILE
        GTJFN%                   ;GET A JFN FOR THE PRINTER
         ERJMPS JFNERR           ;ERROR, PRINT ERROR MESSAGE
        MOVEM T1,LPTJFN          ;REMEMBER PRINTER JFN
        MOVX T2,FLD(7,OF%BSZ)+OF%WR ;7-BIT BYTES,
                                 ;WRITE ACCESS WANTED
```



```
        OPENF%                   ;OPEN THE PRINTER FOR OUTPUT
         ERJMPS OPNERR           ;ERROR, PRINT ERROR MESSAGE

;NOW OUTPUT THE TEXT THAT WAS INPUT FROM THE TERMINAL

        HRROI T2,BUFFER          ;GET POINTER TO TEXT
                                 ;(PRINTER JFN STILL IN T1)
        MOVN T3,COUNT            ;GET NUMBER OF CHARS TO OUTPUT
        SOUT%                    ;OUTPUT STRING OF CHARS TO
                                 ;THE PRINTER
         ERJMPS DATERR           ;ERROR, PRINT ERROR MESSAGE
        TMSG <
OUTPUT HAS BEEN SENT TO THE PRINTER...
>                                ;OUTPUT CONFIRMATION MESSAGE
        MOVE T1,LPTJFN           ;GET PRINTER JFN
        CLOSF%                   ;CLOSE IT
         ERJMPS DATERR           ;UNEXPECTED ERROR, PRINT ERROR MESSAGE
        HALTF%                   ;FINISHED
        JRST START               ;IF CONTINUED, GO BACK TO START

;ERROR ROUTINES

JFNERR: TMSG<
? COULD NOT GET A JFN FOR THE PRINTER
>
        HALTF%
        JRST START               ;IF CONTINUED, GO BACK TO START

OPNERR: TMSG<
? COULD NOT OPEN THE PRINTER FOR OUTPUT
>
        HALTF%
        JRST START               ;IF CONTINUED, GO BACK TO START

DATERR: TMSG<
? DATA ERROR DURING OUTPUT TO PRINTER
>
        HALTF%
        JRST START               ;IF CONTINUED, GO BACK TO START

        END START
```

CHAPTER 4

USING THE SOFTWARE INTERRUPT SYSTEM

4.1 OVERVIEW

Program execution usually occurs in a sequential manner, where instructions are executed one after another. But sometimes a program must be able to receive asynchronous signals from terminals, the monitor, or other programs, or as a result of its own execution. By using the software interrupt system, the user can specify conditions that will cause his program to deviate from its sequential method of execution.

An interrupt is defined as a break in the normal flow of control during a program's execution. The break, or interrupt, is caused by the occurrence of a prespecified condition. By specifying the conditions that can cause an interrupt, the program has the capability of dynamically responding to external events and error conditions and of generating requests for services. Because the program can respond to special conditions as they occur, it does not have to explicitly and repeatedly test for them. In addition, the program's execution is faster because the program does not have to include a special test after the possible occurrence of the condition.

When an interrupt occurs, the system transfers control from the main program sequence to a previously-specified routine that will process the interrupt. After the routine has completed its processing of the interrupt, the system can transfer control back to the program at the point it was interrupted, and execution can continue. See Figure 4-1.

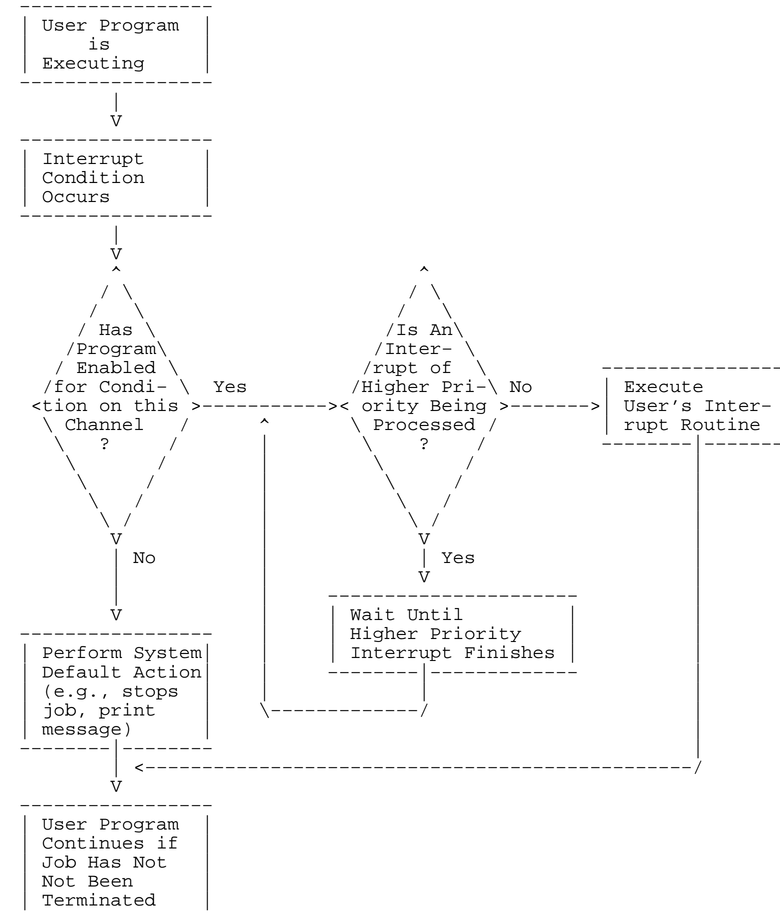


Figure 4-1: Basic Operational Sequence of the Software Interrupt System

4.2 INTERRUPT CONDITIONS

Conditions that cause the program to be interrupted when the interrupt system is enabled are:

1. Conditions generated when specific terminal keys are typed. There are 36 possible codes; each one specifies the particular terminal character or condition on which an interrupt is to be initiated. Refer to Table 4-2 for the possible codes.

2. Invalid instructions (for example, I/O instructions given in user mode) or privileged monitor calls issued by a non privileged user.

3. Memory conditions, such as illegal memory references.

4. Arithmetic processor conditions, such as arithmetic overflow or underflow.

5. Certain file or device conditions, such as end of file.

6. Program-generated software interrupts.

7. Termination of an inferior process.

8. System resource unavailability.

9. Interprocess communication (IPCF) and Enqueue/Dequeue interrupts.

4.3 SOFTWARE INTERRUPT CHANNELS AND PRIORITIES

Each condition is associated with one of 36 software interrupt channels. Most conditions are permanently assigned to specific channels; however, the user's program can associate some conditions (for example, conditions generated by specific terminal keys) to any one of the assignable channels. (Refer to Table 4-1 for the channel assignments.) When the condition associated with a channel occurs, and that channel has been activated, an interrupt is generated. Control can then be transferred to the routine responsible for processing interrupts on that channel.

The user program assigns each channel to one of three priority levels. Priority levels allow the occurrence of some conditions to suspend the processing of other conditions. The levels are referred to as level 1, 2, or 3 with level 1 having the highest priority. Level 0 is not a legal priority level.[1]

Table 4-1: Software Interrupt Channel Assignments

Channel	Symbol	Meaning
0-5		Assignable by user program
6	.ICAOV	Arithmetic overflow
7	.ICFOV	Arithmetic floating point overflow
8		Reserved for Digital
9	.ICPOV	Pushdown list (PDL) overflow*
10	.ICEOF	End of file condition
11	.ICDAE	Data error file condition*
12	.ICQTA	Disk quota exceeded
13-14		Reserved for Digital
15	.ICILI	Illegal instruction*
16	.ICIRD	Illegal memory read*
17	.ICIWR	Illegal memory write*
18		Reserved for Digital
19	.ICIFT	Inferior process termination
20	.ICMSE	System resources exhausted*
21		Reserved for Digital
22	.ICNXP	Nonexistent page reference
23-35		Assignable by user program

----

[1] If an interrupt is generated in a process where the priority level is 0, the system considers that the process is not prepared to handle the interrupt. The process is then suspended or terminated according to the setting of bit 17 (SC%FRZ) in its capability word.

* These channels (called panic channels) cannot be completely deactivated. An interrupt generated on one of these channels terminates the process if the channel is not activated.

The software interrupt system processes interrupts on activated channels only, and each channel can be activated and deactivated independently of other channels. When activated, the channel can generate an interrupt for its associated priority level. An interrupt for any priority level is initiated only if there are no interrupts in progress for the same or higher priority levels. If there are, the system remembers the interrupt request and initiates it after all equal or higher priority level interrupts finish. This means that a higher priority level request can suspend a routine processing a lower level interrupt. Thus, the user must be concerned with several items when he assigns his priority levels. He must consider 1) when one interrupt request can suspend the processing of another and 2) when the processing of a second interrupt cannot be deferred until the completion of the first.

4.4 SOFTWARE INTERRUPT TABLES

To process interrupts, the user includes, as part of his program, special service routines for the channels he will be using. He must then specify the addresses of these routines to the system by setting up a channel table. In addition, the user must also include a priority level table as part of his program. Finally, he must declare the addresses of these tables to the system.

4.4.1 Specifying the Software Interrupt Tables

Before using the software interrupt system, the user's program must set up the contents of the channel table and the priority level table. The program must then specify their addresses with either the SIR% or XSIR% monitor calls.

These calls are similar, but their differences are important. The SIR% call can be used in single-section programs, but the XSIR% call must be used in programs that use more than one section of memory. The SIR% call works in non-zero sections only if the tables are in the same section as the code that makes the call. The code that causes the interrupt must also be in that section, as must the code that processes the interrupt. Because of the limitations of the SIR% call, you should use the XSIR% call.

The SIR% monitor call accepts two words of arguments: the identifier for the program (or process) in AC1, and the table addresses in AC2. Refer to Section 5.3 for the description of process identifiers.

The following example shows the use of the SIR% call.

```
     MOVEI 1,.FHSLF           ;identifier of current process
     MOVE 2,[LEVTAB,,CHNTAB]  ;addresses of the tables
     SIR%
```

The XSIR% call accepts the following arguments: in AC1, the identifier of the process for which the interrupt channel tables are to be set; in AC2, the address of the argument block.

The argument block is a three-word block that has the following format:

```
!=====================================================!
!   Length of the argument block, including this word  !
!-----------------------------------------------------!
!          Address of the interrupt level table        !
!-----------------------------------------------------!
!             Address of the channel table             !
!=====================================================!
```

Control always returns to the user's program at the instruction following the SIR% and XSIR% calls. If the call is successful, the table addresses are stored in the monitor. If the call is not successful, an illegal instruction trap is generated.

Any changes made to the contents of the tables after the XSIR% or SIR% calls have been executed will be in effect at the time of the next interrupt.

4.4.2 Channel Table

The channel table, CHNTAB,[2] contains a one-word entry for each channel; thus, the table has 36 entries. Each entry corresponds to a particular channel, and each channel is associated at any given time with only one interrupt condition. (Refer to Table 4-1 for the interrupt conditions associated with each channel.)

The CHNTAB table is indexed by the channel number (0 through 35). The general format, for use with the XSIR% and XRIR% monitor calls, can be used in any section of memory. The left half of each entry contains the priority level (1, 2, or 3) in bits 0-5 (SI%LEV) to which the channel is assigned. Bits 6-35 (SI%ADR) of each entry contain the starting address of the routine to process interrupts generated on

----

[2] The user can call his priority channel table any name he desires; however, it is good practice to call it CHNTAB.

that channel. If a particular channel is not used, the corresponding entry in the channel table should be zero.

In the older format, for use with the SIR% and RIR% calls by any single-section program, the left half of each word contains the priority level (1, 2, or 3) for that channel. The right half contains the address of the interrupt routine that will handle interrupts on that channel.

The following example is for use with the XSIR% monitor call.

```
CHNTAB: FLD(2,SI%LEV)+FLD(CHN0SV,SI%ADR)        ;channel 0
        FLD(2,SI%LEV)+FLD(CHN1SV,SI%ADR)        ;channel 1
        FLD(2,SI%LEV)+FLD(CHN2SV,SI%ADR)        ;channel 2
        FLD(2,SI%LEV)+FLD(CHN3SV,SI%ADR)        ;channel 3
        0                                       ;channel 4
        0                                       ;channel 5
        FLD(1,SI%LEV)+FLD(APRSRV,SI%ADR)        ;channel 6
        0                                       ;channel 7
        0                                       ;channel 8
        FLD(1,SI%LEV)+FLD(STKSRV,SI%ADR)        ;channel 9
        0                                       ;channel 10
         .                                        .
         .                                        .
         .                                        .
        0                                       ;channel 35
```

In this example, channels 0 through 3 are assigned to priority level 2, with the interrupt routine at CHN0SV servicing channel 0, the routine at CHN1SV servicing channel 1, the routine at CHN2SV servicing channel 2, and the routine at CHN3SV servicing channel 3. Channels 6 and 9 are assigned to priority level 1, with the routine at APRSRV servicing channel 6 and the routine at STKSRV servicing channel 9. All remaining channels are not assigned.

4.4.3 Priority Level Table

The priority level table, LEVTAB,[3] The priority level table, LEVTAB, [3] is a three-word table, containing a one-word entry for each of the three priority levels. In the general form, each word contains the 30-bit address of the first word of the two-word block in the process address space. The block addressed by word n of LEVTAB is used to store the global PC flags and address when an interrupt of level n+1 occurs.

The PC flags are stored in the first word of the PC block, and the PC

----

[3] The user can call his priority level table any name he desires; however, it is good practice to call it LEVTAB.

address is stored in the second. This form of the table must be used with the XSIR% and XRIR% monitor calls, and can be used in any section.

The older form of the interrupt level table can be used in any single-section program, and must be used with the SIR% and RIR% calls. This table also contains three words, indexed by the priority level minus 1. Each word contains zero in the left half, and the 18-bit address of the word in which to store the one-word section-relative PC in the right half. This address is assumed to be in the same program section that contained the SIR% monitor call. (For more information see Chapter 8.) The system must save the value of the program counter so that it can return control at the appropriate point in the program once the interrupt routine has completed processing an interrupt. If a particular priority level is not used, its corresponding entry in the level table should be zero.

The following is a sample of a level table.

```
LEVTAB:  0,,PCLEV1       ;Addresses to save PC for interrupts
         0,,PCLEV2       ;occurring on priority levels 1 and 2.
         0,,0            ;No priority level 3 interrupts are
                         ;planned
```

4.5 ENABLING THE SOFTWARE INTERRUPT SYSTEM

Once the interrupt tables have been set up and their addresses defined with the XSIR% monitor call, the user's program must enable the interrupt system. When the interrupt system is enabled, interrupts that occur on activated channels are processed by the user's interrupt routines. When the interrupt system is disabled, the monitor processes interrupts as if the channels for these interrupts were not activated.

The EIR% monitor call, used to enable the system, accepts one argument: the identifier for the process in AC1.

```
MOVEI 1,.FHSLF          ;identifier of current process
EIR%
```

Control always returns to the instruction following the EIR call.

4.6 ACTIVATING INTERRUPT CHANNELS

Once the software interrupt system is enabled, the channels on which interrupts can occur must be activated (refer to Table 4-1 for the channel assignments). The channels to be activated have a nonzero entry in the appropriate word in the channel table.

The AIC% monitor call activates one or more of the 36 interrupt channels. This call accepts two words of arguments - the identifier for the process in AC1, and the channels to be activated in AC2.

The channels are indicated by setting bits in AC2. Setting bit n indicates that channel n is to be activated. The AIC% call activates only those channels for which bits are set.

```
MOVEI 1,.FHSLF                    ;identifier of current process
MOVE 2,[1B<.ICAOV>+1B<.ICPOV>]    ;activate channels 6 and 9
AIC%
```

Control always returns to the instruction following the AIC% call.

Some channels, called panic channels, cannot be deactivated by disabling the channel or the entire interrupt system. (Refer to Table 4-1 for these channels.) This is because the occurrence of the conditions associated with these channels cannot be completely ignored by the monitor.

If one of these conditions occurs, an interrupt is generated whether the channel is activated or not. If the channel is not activated, the process is terminated, and usually a message is output before control returns to the monitor. If the channel is activated, control is given to the user's interrupt routine for that channel.

4.7 GENERATING AN INTERRUPT

A process generates an interrupt by producing a condition for which an interrupt channel is enabled, such as arithmetic overflow, or by using the IIC% monitor call. This call can generate an interrupt on any of the 36 interrupt channels of the process the calling process specifies. See Section 5.10 for a description of the IIC% call.

4.8 PROCESSING AN INTERRUPT

When a software interrupt occurs on a given priority level, the monitor stores the current program counter (PC) word in the address indicated in the priority level table (refer to Section 4.4.3). The monitor then transfers control to the interrupt routine associated with the channel on which the interrupt occurred. The address of this routine is specified in the channel table (refer to Section 4.4.2).

Since the user's program cannot determine when an interrupt will occur, the interrupt routine must preserve the state of the program so the program can be resumed properly. First, the routine stores the contents of any user accumulators for use while processing the interrupt. After the accumulators are saved, the interrupt routine processes the interrupt.

Occasionally, an interrupt routine may need to alter locations in the main section of the program. For example, a routine may change the stored PC word to resume execution at a location different from where the interrupt occurred. Or it may alter a value that caused the interrupt. It is important that care be used when writing routines that alter data because any changes will remain when control is returned to the main program. For example, if data is inadvertently stored in the PC word, return to the main section of the program would be incorrect when the system attempted to use the word as the value of the program counter.

If a higher-priority interrupt occurs during the execution of an interrupt routine, the execution of the lower-priority routine is suspended. The value of its program counter is stored at the location indicated in the priority level table for the new interrupt. When the routine for this new interrupt is completed, the suspended routine resumes.

If an interrupt of the same or lower priority occurs during the execution of a routine, the monitor holds the interrupt until all higher or equal level interrupts have been processed.

The system considers the user's program unable to process an interrupt on an activated channel if any of the following is true:

1. The priority level associated with the channel is 0.
2. The program has not defined its interrupt tables by executing an XSIR% or SIR% monitor call.
3. The process has not enabled the interrupt system by executing an EIR% monitor call, and the channel on which the interrupt occurs is a panic channel.

In any of these cases, an interrupt on a panic channel terminates the user's program. All other interrupts are ignored.

4.8.1 Dismissing an Interrupt

Once the processing of an interrupt is complete, the interrupt routine should restore the user accumulators to their initial values. Then it should return control to the interrupted code by using the DEBRK% monitor call. This call restores the PC word and resumes the program. The call has no arguments, and must be the last statement in the interrupt routine.

If the interrupt-processing routine has not changed the PC of the user's program, the DEBRK% call restores the program to the same state

the program was in just before the interrupt occurred. If the program was interrupted while waiting for I/O to complete, for example, the program will again be waiting for I/O to complete when it resumes execution after the DEBRK% call.

If the PC word was changed, the program resumes execution at the new PC location. The state of the program is unchanged.

4.9 TERMINAL INTERRUPTS

The user's program can associate channels 0 through 5 and channels 24 through 35 with occurrences of various conditions, such as the occurrence of a particular character typed at the terminal or the receipt of an IPCF message. This section discusses terminal interrupts; refer to Chapters 6 and 7 for other types of assignable interrupts.

There are 36 codes used to specify terminal characters or conditions on which interrupts can be initiated. These codes, along with their associated conditions, are shown in Table 4-2.

Table 4-2: Terminal Codes and Conditions

Code	Symbol	Character or Condition
0	.TICBK	CTRL/@ or break
1	.TICCA	CTRL/A
2	.TICCB	CTRL/B
3	.TICCC	CTRL/C
4	.TICCD	CTRL/D
5	.TICCE	CTRL/E
6	.TICCF	CTRL/F
7	.TICCG	CTRL/G
8	.TICCH	CTRL/H
9	.TICCI	CTRL/I
10	.TICCJ	CTRL/J

11	.TICCK	CTRL/K
12	.TICCL	CTRL/L
13	.TICCM	CTRL/M
14	.TICCN	CTRL/N
15	.TICCO	CTRL/O
16	.TICCP	CTRL/P
17	.TICCQ	CTRL/Q
18	.TICCR	CTRL/R
19	.TICCS	CTRL/S
20	.TICCT	CTRL/T
21	.TICCU	CTRL/U
22	.TICCV	CTRL/V
23	.TICCW	CTRL/W
24	.TICCX	CTRL/X
25	.TICCY	CTRL/Y
26	.TICCZ	CTRL/Z
27	.TICES	ESC key
28	.TICRB	Delete (or rubout) key
29	.TICSP	Space
30	.TICRF	Dataset carrier off
31	.TICTI	Typein
32	.TICTO	Typeout
33	.TITCE	Two-character escape sequence
34-35		Reserved

To cause terminal interrupts to be generated, the user's program must assign the desired terminal code to one of the assignable channels.

The ATI% monitor call is used to assign this code. This call accepts one word of arguments: the terminal code in the left half of AC1 and the channel number in the right half.

```
MOVE 1,[.TICCE,,INTCH1]  ;assign CTRL/E to channel INTCH1
ATI%
```

Control always returns to the instruction following the ATI% call. If the current job is not attached to a terminal (there is no terminal controlling the job), the terminal code assignments are remembered; they will be in effect when a terminal is attached.

The monitor handles the receipt of a terminal interrupt character in either immediate mode or deferred mode. In immediate mode, the terminal character causes the system to initiate an interrupt as soon as the user types the character (that is, as soon as the system receives it). In deferred mode, the terminal character is placed in either immediate mode or deferred mode. In immediate mode, the terminal character causes the system to initiate an interrupt as soon as the user types the character (as soon as the system receives it). In deferred mode, the terminal character is placed in the input stream in sequence with other characters of the input, unless two of the same character are typed in succession. In this case, an interrupt occurs at the time the second one is typed. If only one character enabled in deferred mode is typed, the system initiates an interrupt only when the program attempts to read the character. Deferred mode allows interrupt actions to occur in sequence with other actions specified in the input (for example, when characters are typed ahead of the time that the program actually requests them). In either mode, the character is not passed to the program as data. The system assumes that interrupts are to be handled immediately unless a program has issued the STIW% (Set Terminal Interrupt Word) monitor call. (Refer to <u>TOPS-20 Monitor Calls Reference Manual</u> for a description of this call.)

4.10 ADDITIONAL SOFTWARE INTERRUPT MONITOR CALLS

Additional monitor calls are available that allow the user's program to check and to clear various parts of the software interrupt system. Also, there is a call useful for interprocess communication (refer to the IIC% call in Section 5.10).

4.10.1 Testing for Enablement

The SKPIR% monitor call tests the software interrupt system to see if it is enabled. The call accepts in AC1 the identifier of the process. After execution of the call, control returns to the next instruction if the system is off, and to the second instruction if the system is on.

```
MOVEI 1,.FHSLF          ;identifier of current process
SKPIR%                  ;test interrupt system
 return                 ;system is off
return                  ;system is on
```

4.10.2 Obtaining Interrupt Table Addresses

The RIR% and XRIR% monitor calls obtain the channel and priority level table addresses for a process. These calls are useful when several routines in one process want to share the interrupt tables.

4.10.2.1 The RIR% Monitor Call - The RIR% monitor call can be used in any section of memory, but is only useful for obtaining table addresses if those tables are in the same section of memory as the code that makes the call. Furthermore, it can only obtain table addresses that have been set by the SIR call.

The call accepts the identifier of the process in AC1. It returns the table addresses in AC2. The left half of AC2 contains the section-relative address of the priority level table, and the right half contains the section-relative address of the channel table. If the process has not set the table addresses with the SIR% monitor call, AC2 contains zero.

Control always returns to the instruction following the RIR% call.

The following example shows the use of the RIR% call.

```
MOVEI 1,.FHSLF          ;identifier of current process
RIR%                    ;return the table addresses
```

4.10.2.2 The XRIR% Monitor Call - This call obtains the addresses of the interrupt tables defined for a process. The tables can be in any section of memory. The code that makes the call can also be in any section. This call can only obtain addresses that have been set by the XSIR% call.

The call accepts the identifier of the process in AC1, and the address of the argument block in AC2. The argument block is three words long, word zero must contain the number 3. The call returns the addresses into words one and two. The block has the following format:


```
!=======================================================!
!   Length of the argument block, including this word   !
!-------------------------------------------------------!
!          Address of the interrupt level table         !
!-------------------------------------------------------!
!              Address of the channel table             !
!=======================================================!
```

Control always returns to the instruction following the XRIR% call. If the process has not set the table addresses with the XSIR% monitor call, words one and two of the argument block contain zero.

4.10.3 Disabling the Interrupt System

The DIR% monitor call disables the software interrupt system for the process. It accepts the identifier of the process in AC1.

```
MOVEI 1,.FHSLF          ;identifier of current process
DIR%                    ;disable system
```

Control always returns to the instruction following the DIR% call.

If interrupts occur while the interrupt system is disabled, they are remembered until the system is reenabled. At that time, the interrupts take effect unless an intervening CIS% monitor call (refer to Section 4.10.6) has been issued.

Software interrupts assigned to panic channels are not completely disabled by the DIR% call. These interrupts terminate the process, and the superior process is notified if it has enabled channel .ICIFT. In addition, if the terminal code for CTRL/C (.TICCC) is assigned to a channel, it causes an interrupt that cannot be disabled by the DIR% call. However, the CTRL/C interrupt can be disabled by deactivating the channel assigned to the CTRL/C terminal code.

4.10.4 Deactivating a Channel

The DIC% monitor call is used to deactivate interrupt channels. The call accepts two words of arguments: the process identifier in AC1, and the channels to be deactivated in AC2. Setting bit n in AC2 indicates that channel n is to be deactivated.

```
MOVEI 1,.FHSLF                  ;identifier of current process
MOVE 2,[1B<.ICAOV>+1B<.ICPOV>]  ;deactivate channels 6 and 9
DIC%
```

Control always returns to the instruction following the DIC% call.

When a channel is deactivated, interrupt requests for that channel are ignored except for interrupts generated on panic channels (refer to Section 4.6).

4.10.5 Deassigning Terminal Codes

The DTI% monitor call deassigns a terminal code. This call accepts one argument word: the terminal code in AC1.

```
     MOVEI 1,.TICCE           ;deassign CTRL/E
     DTI%
```

Control always returns to the instruction following the DTI% call. This monitor call is ignored if the specified terminal code has not been defined by the current job.

4.10.6 Clearing the Interrupt System

The CIS% monitor call clears the interrupt system for the current process. This call clears interrupts in progress and all waiting interrupts. This call requires no arguments, and control always returns to the instruction following the CIS call. The RESET% monitor call (refer to Section 2.6.1) performs these same actions as part of its initializing procedures.

4.11 SUMMARY

To use the software interrupt system, the user's program must:

1. Supply routines that will process the interrupts.

2. Set up a channel table containing the addresses of the routines (refer to Section 4.4.2) and a priority level table containing the addresses for storing the program counter (PC) values (refer to Section 4.4.3).

3. Specify the addresses of the tables with the XSIR% monitor call (refer to Section 4.4.3).

4. Enable the software interrupt system with the EIR% monitor call (refer to Section 4.5).

5. Activate the desired channels with the AIC% monitor call (refer to Section 4.6).

4.12 SOFTWARE INTERRUPT EXAMPLE

This program copies one file to another. It accepts the input and output filenames from the user. The end of file is detected by a software interrupt, and CTRL/E is enabled as an escape character.

```
        TITLE SOFTWARE INTERRUPT EXAMPLE
        SEARCH MONSYM
        SEARCH MACSYM
        .REQUIRE SYS:MACREL

        STDAC.                  ;DEFINE STANDARD ACs
        INTCH1=1

START:  RESET%                  ;RELEASE FILES, ETC.
        XHLLI T1,EOFINT         ;GET CURRENT PROCESS SECTION NUMBER
        HLLZS T1                ;ISOLATE SECTION NUMBER ONLY
        IORM T1,CHNTAB+INTCH1   ; AND ADD IT TO SERVICE ROUTINE
        IORM T1,CHNTAB+.ICEOF   ;ADDRESSES FOR OUR ROUTINES
        IORM T1,LEVTAB+1        ; AND LEVTAB
        MOVEI T1,.FHSLF         ;CURRENT PROCESS
        MOVEI T2,3              ;NUMBER OF WORDS IN ARG BLOCK
        MOVEM T2,ARGBLK         ;PUT NUMBER IN WORD ZERO
        XMOVEI T2,LEVTAB        ;GLOBAL ADDRESS OF LEVEL TABLE
        MOVEM T2,ARGBLK+1       ;MOVE IT TO ARGBLK WORD ONE
        XMOVEI T2,CHNTAB        ;GLOBAL ADDRESS OF CHANNEL TABLE
        MOVEM T2, ARGBLK+2      ;MOVE IT TO ARGBLK WORD TWO
        XMOVEI T2,ARGBLK        ;GLOBAL ADDRESS OF ARGUMENT BLOCK
        XSIR%
        EIR%                    ;ENABLE SYSTEM
        MOVE T2,[1B<INTCH1>+1B<.ICEOF>] ;ACTIVATE CHANNELS
        AIC%
        MOVE T1,[.TICCE,,INTCH1] ;ASSIGN CTRL/E TO CHANNEL 1
        ATI%
GETIF:  TMSG <INPUT FILE: >
        MOVX T1,GJ%OLD+GJ%MSG+GJ%CFM+GJ%FNS+GJ%SHT
        MOVE T2,[.PRIIN,,.PRIOU]
        GTJFN%                  ;GET FILENAME FROM USER
         ERJMP ERROR1
        MOVEM T1,INJFN
GETOF:  TMSG <OUTPUT FILE: >
        MOVX T1,GJ%FOU+GJ%MSG+GJ%CFM+GJ%FNS+GJ%SHT
        MOVE T2,[.PRIIN,,.PRIOU]
        GTJFN%                  ;GET FILENAME FROM USER
         ERJMP ERROR2
        MOVEM T1,OUTJFN
```

```
OPNIF:  MOVE T1,INJFN
        MOVX T2,FLD(7,OF%BSZ)+OF%RD
        OPENF%                  ;OPEN INPUT FILE
         ERJMP ERROR3
OPNOF:  MOVE T1,OUTJFN
        MOVX T2,FLD(7,OF%BSZ)+OF%WR
        OPENF%                  ;OPEN OUTPUT FILE
         ERJMP ERROR3
CPYBYT: MOVE T1,INJFN
        BIN%                    ;READ INPUT BYTE
        MOVE T1,OUTJFN
        BOUT%                   ;WRITE OUTPUT BYTE
        JRST CPYBYT             ;LOOP UNTIL EOF
DONE:   MOVE T1,INJFN
        CLOSF%                  ;CLOSE INPUT FILE
         JFCL
        MOVE T1,OUTJFN
        CLOSF%                  ;CLOSE OUTPUT FILE
         JFCL
        HALTF%
;ROUTINE TO HANDLE ^E - ABORTS OPERATION

CTRLE:  MOVEI T1,.PRIOU
        CFOBF%                  ;CLEAR OUTPUT BUFFER
        TMSG <ABORTED.>         ;INFORM USER
        CIS%                    ;CLEAR SYSTEM
        JRST START

;ROUTINE TO HANDLE EOF - COMPLETES OPERATION NORMALLY

EOFINT: MOVEM T1,INTAC1         ;SAVE ACs
        XMOVEI T1,DONE          ;CHANGE PC
        MOVEM T1,PC2+1          ;TO DONE
        MOVE T1,INTAC1          ;RESTORE ACs
        DEBRK%                  ;DISMISS INTERRUPT

;LEVEL TABLE
LEVTAB: 0
        PC2
        0
PC2:    BLOCK 2
```



```
;CHANNEL TABLE
CHNTAB: 0
        FLD(2,SI%LEV)!FLD(CTRLE,SI%ADR)
        REPEAT ^D8,<0>
        FLD(2,SI%LEV)!FLD(EOFINT,SI%ADR)
        REPEAT ^D25,<0>
ARGBLK: BLOCK 3
INJFN:  BLOCK 1
OUTJFN: BLOCK 1
INTAC1: BLOCK 1
ERROR1: TMSG <
?INVALID FILE SPECIFICATION>
        HALTF%
ERROR2: TMSG <
?INVALID FILE SPECIFICATION>
        HALTF%
ERROR3: TMSG <
?CANNOT OPEN FILE>
        HALTF%
        LIT
        END START
```

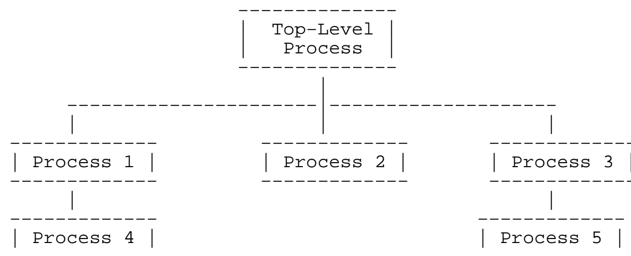
CHAPTER 5

PROCESS STRUCTURE

As stated in Chapter 1, the TOPS-20 operating system allows each job to have multiple processes that can run simultaneously. Each process has its own environment called its address space. Associated with the environment is the program counter (PC) of the process and a well-defined relationship with other processes in the job. In TOPS-20, the term fork is synonymous with the term process.

The TOPS-20 operating system schedules the running of processes, not entire jobs. A process can be scheduled independent of other processes because it has a definite existence: its beginning is the time at which it is created, and its end is the time at which it is killed. At any point in its existence, a process can be described by its state, which is represented by a status word and a PC word (refer to Section 5.9).

The relationships among processes in a job are shown in the diagram below. Each process has one immediate superior process (except for the top-level process) and can have one or more inferior processes. Two processes are parallel if they have the same immediate superior. A process can create an inferior process but not a parallel or superior process.



Process 1 is the superior process of process 4, and process 3 is the superior of process 5. Processes 4 and 5 are the inferiors of processes 1 and 3, respectively. Process 2 has no inferior process. Processes 1, 2 and 3 are parallel because they have the same superior process (the top-level process). Processes 4 and 5, although at the same depth in the structure, are not parallel because they do not have the same superior process. Process 1 created process 4 but could not have created any other process shown in the structure above.

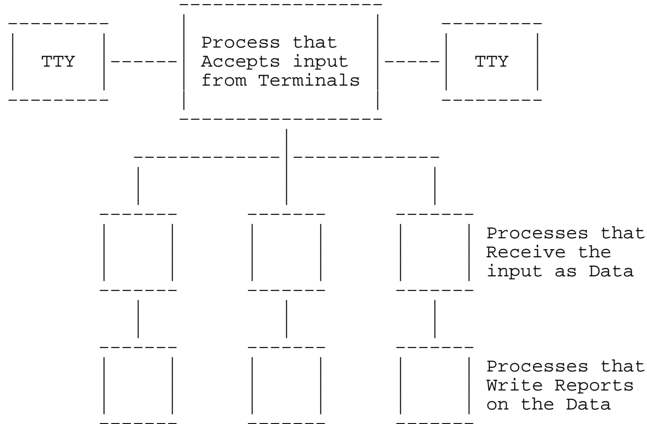
5.1 USES FOR MULTIPLE PROCESSES

A multiple-process job structure allows:

1. One job to have more than one program runnable at the same time. These programs can be independent programs, each one compiled, debugged, and loaded separately. Each program can then be placed in a separate process. These processes can be parallel to each other, but are inferior to the main process that created them. This use allows parallel execution of the individual programs.

2. One process to wait for an event to occur (for example, the completion of an I/O operation) while another process continues its computations. Communication between the two processes is such that when the event occurs, the process that is computing can be notified via the software interrupt system. This use allows two processes within a job to overlap I/O with computations.

One application of a multiple-process job structure is the following situation: a superior process is responsible for accepting input from various terminals. After receiving this input, the process sends it to various inferior processes as data. These inferior processes can then initiate other processes, for example, to write reports on the data that was received.



Another application is that used for the user interface on the DECSYSTEM-20. On the DECSYSTEM-20, the top-level process in the job structure is the Command Language. This process services the user at the terminal by accepting input. When the user runs a program (for example, MACRO, FORTRAN), the Command Language process creates an inferior process, places the requested program in it, and executes it. The Command Language can then wait for an event to occur, either from the program or from the user. An event from the program can be its completion, and an event from the user can be the typing of a certain terminal key (CTRL/C, for example).

5.2 PROCESS COMMUNICATION

A process can communicate with or control other processes in the system in several ways:

- o direct process control
- o software interrupts
- o IPCF and ENQ/DEQ facilities
- o memory sharing

5.2.1 Direct Process Control

A process can create and control other processes inferior to it within the job structure. The superior process can cause the inferior process to begin execution and then to suspend and later resume execution. After the inferior process has completed its tasks, the superior process can delete the inferior from the job structure.

Some of the monitor calls used for direct process control are: CFORK%, to create a process; SFORK%, to start a process; WFORK%, to wait for a process to terminate; RFSTS%, to obtain the status of a process; and KFORK%, to delete a process. Refer to the TOPS-20 Monitor Calls Reference Manual for descriptions of additional monitor calls dealing with process control.

5.2.2 Software Interrupts

The software interrupt facility enables a process to receive asynchronous signals from other processes, the system, or the terminal user or to receive signals as a result of its own execution. For example, a superior process can enable the interrupt system so that it receives an interrupt when one of its inferiors terminates. In addition, processes within a job structure can explicitly generate interrupts to each other for communication purposes.

Some of the monitor calls used when communication occurs via the software interrupt system are: SIR%, to specify the interrupt tables; EIR%, to enable the interrupt system; AIC%, to activate the interrupt channels; and IIC%, to initiate an interrupt on a channel. Refer to Chapter 4 and Section 5.10 for more information.

5.2.3 IPCF and ENQ/DEQ Facilities

The Inter-Process Communication Facility (IPCF) enables processes and jobs to communicate by sending and receiving informational messages. The MSEND% call is used to send a message, the MRECV% call is used to receive a message, and the MUTIL% call is used to perform utility functions. Refer to Chapter 7 for descriptions of these calls.

The ENQ/DEQ facility allows cooperating processes to share resources and facilitates dynamic resource allocation. The ENQ% call is used to obtain a resource, the DEQ% call is used to release a resource, and the ENQC% call is used to obtain status about a resource. Refer to Chapter 6 for descriptions of these calls.

5.2.4 Memory Sharing

Each page or section in a process' address space is either private to the process or shared with other processes. Pages are shared among processes when the same page is represented in more than one process' address space. This means that two or more processes can identify and use the same page of physical storage. Even when several processes have identified the same page, each process can have a different access to that page, such as access to read or write that page.

A type of page access that facilitates sharing is the copy-on-write access. A page with this access remains shared as long as all processes read the page. As soon as a process writes to the page, the system makes a private copy of the page for the process doing the writing. Other processes continue to read and execute the original page. This access provides the capability of sharing as much as possible but still allows the process to change its data without changing the data of other processes. A monitor call used when sharing memory is PMAP%. Refer to Section 5.6.2 for more information.

5.3 PROCESS IDENTIFIERS

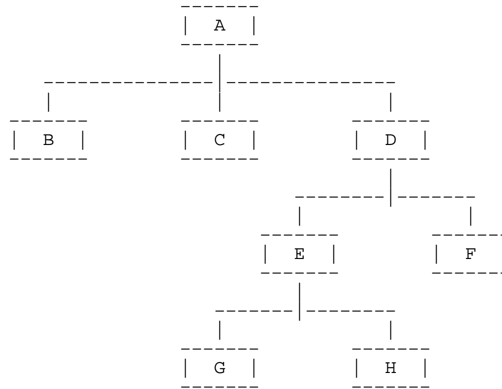
In order for processes to communicate with each other, a process must have an identifier, or handle, for referencing another process. When a process creates an inferior process, it is given a handle on that inferior. This handle is a number in the range 400001 to 400777 and is meaningful only to the process to which it is given (that is, to the superior process). For example, if process A creates process B, process A is given a handle (for example, 400003) on process B. Process A then specifies this handle when it uses monitor calls that refer to process B. However, process B is not known by this handle to any other process in the structure, including itself. The handle 400003 may in fact be known to process B, but it would describe a process inferior to process B. For this reason, process handles are sometimes called "relative fork handles" because they are relative to the process that created them.

There are several standard process handles that are never assigned by the system but have a specific meaning when used by any process in the structure. These handles are used when a process needs to communicate with a process other than its immediate inferior or with multiple processes at once. These handles are described in Table 5-1.

Table 5-1: Process Handles

Number	Symbol	Meaning
400000	.FHSLF	The current process (or self).
400000+n		Process n, relative to the current process.
200000	FH%EPN	Extended page number (see PM%EPN in PMAP%). When used in conjunction with the above two forms, this bit indicates that addresses and/or page numbers are interpreted as absolute, not relative to the PC section of the program executing the JSYS. This bit has no meaning for programs that do not use extended addressing.
-1	.FHSUP	The immediate superior of the current process.
-2	.FHTOP	The top-level process in the job structure.
-3	.FHSAI	The current process and all of its inferiors.
-4	.FHINF	All of the inferiors of the current process.
-5	.FHJOB	All processes in the job structure.

Consider the job structure below.



The following indicates the specific process or processes being referenced if process E gives the handle:

```
.FHSLF    refers to process E
.FHSUP    refers to process D
.FHTOP    refers to process A
.FHSAI    refers to processes E, G, and H
.FHINF    refers to processes G and H
.FHJOB    refers to processes A through H
```

The process must have the appropriate capability enabled in its capability word to use the handles .FHSUP, .FHTOP, and .FHJOB (refer to Section 5.5.1).

Process E can reference one of its inferiors (for example, G) with the handle it was given when it created the inferior. Process E can reference other processes in the structure (for example, F) by executing the GFRKS% monitor call to obtain a handle on the desired process. Refer to the TOPS-20 Monitor Calls Reference Manual for a description of the GFRKS% call.

5.4 OVERVIEW OF MONITOR CALLS FOR PROCESSES

Monitor calls exist for creating, loading, starting, suspending, resuming, interrupting, and deleting processes. When a process is created, its address space is assigned, and the process is added to the job structure of the creating process. The contents of its

address space can be specified at the time the process is created or at a later time. The process can also be started at the time it is created. A process remains potentially runnable until it is explicitly deleted or its superior is deleted.

A process may be suspended if one of the following conditions occurs:

1. The process executes an instruction that causes a software interrupt to occur, and it is not prepared to process the interrupt.
2. The process executes the HALTF% monitor call.
3. The superior process requests suspension of its inferior.
4. The superior process is suspended. When a process is suspended, all of its inferior processes are also suspended.
5. A monitor call is trapped. (Refer to TFORK% monitor call in the TOPS-20 Monitor Calls Reference Manual).

5.5 CREATING A PROCESS

A process creates an inferior process by executing the CFORK% (Create Process) monitor call. This monitor call allows the caller to specify the address space, capabilities, initial contents of the ACs, and PC for the inferior process and to start the execution of the inferior.

The CFORK% call accepts two words of arguments in AC1 and AC2.

AC1: characteristics for the inferior in the left half, and PC address for the inferior in the right half.

AC2: address of a 20 (octal) word block containing the AC values for the inferior.

The characteristics for the inferior process are described in Table 5-2.

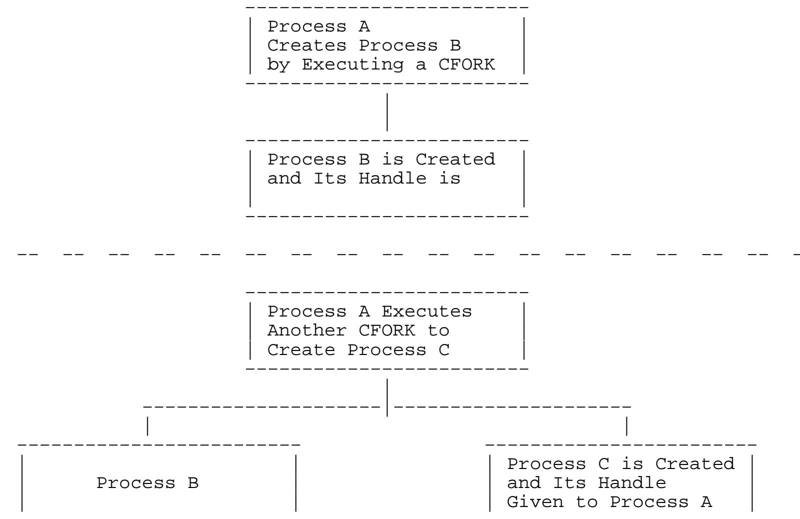
Table 5-2: Inferior Process Characteristic Bits

Bit	Symbol	Meaning
0	CR%MAP	Set the map of the inferior process to the same as the map of the superior (creating) process. This means that the superior and the inferior will share the same address space. Changes made by one process will be seen by the other process. If this bit is not on in the call, the inferior's map will contain all zeros. If desired, the creating process can then use PMAP or GET to add pages to the inferior's map.
1	CR%CAP	Set the capability word of the inferior process to the same as the capability word of the superior process. (Refer to Section 5.5.1 for the description of the capability word.) If this bit is not on in the call, the inferior will have no special capabilities.
2		Reserved for Digital (must be 0).
3	CR%ACS	Set the ACs of the inferior process to the values beginning at the address given in AC2. If this bit is not on in the call, the inferior's ACs will be set to zero, and the contents of AC2 is ignored.
4	CR%ST	Set the PC for the inferior process to the address given in the right half of AC1 and start execution of the inferior. If this bit is not on in the call, the right half of AC1 is ignored, and the inferior is not started. If desired, the creating process can then use SFORK% or XSFRK% to start the newly created process.
18-35	CR%PCV	PC value for inferior process if CR%ST is on.

If execution of the CFORK% call is not successful, the inferior process is not created and an error code is returned, as described in Section 1.2.2.

If execution of the CFORK% call is successful, the inferior process is created and its process handle is returned in the right half of AC1. This handle is then used by the superior process when communicating with its inferior process. The execution of the program in the superior process continues at the second instruction following the CFORK% call. The inferior begins execution at the location contained in bits 18-35 (CR%PCV) if CR%ST is specified.

Assume that process A executes the CFORK% monitor call twice to create two parallel inferior processes. This is represented pictorially below.



Note that process A has been given two handles, one for process B and one for process C. Process A can refer to either of its inferiors by giving the appropriate handle or to both of its inferiors by giving a handle of -4 (.FHINF).

5.5.1 Process Capabilities

When a new process is created, it is given the same capabilities as its superior, or it is given no special capabilities. This is indicated by the setting of the CR%CAP bit in the CFORK% call. The capabilities for a process are indicated by two capability words. The first word indicates if the capability is available to the process, and the second word indicates if the capability is enabled for the process. This second word is the one being set by the CR%CAP bit in the CFORK% call.

Types of capabilities represented in the capability words are job, process, and user capabilities. Each capability corresponds to a particular bit in the capability words and thus can be activated and protected independently of the other capabilities. Refer to the TOPS-20 Monitor Calls Reference Manual for more information on the capability words.

5.6 SPECIFYING THE CONTENTS OF THE ADDRESS SPACE OF A PROCESS

Once a process is created, the contents of its address space can be specified. This can be accomplished in one of three ways. As mentioned in Section 5.5, bit CR%MAP can be set in the CFORK% call to indicate that the address space of the inferior process is to be the same as the address space of the creating process. In addition, the creating process can execute the GET% monitor call to map specified pages from a file into the address space of the inferior process. Finally, the creating process can execute the PMAP% monitor call to map specified pages from another process into the address space of the inferior process.

If the creating process does not specify the contents of the inferior's address space, the address space will be filled with zeros.

5.6.1 GET% Monitor Call

The GET% monitor call gets a save file, copying or mapping it into the process as appropriate. It updates the monitor's data base for the process by copying the entry vector and the list of program data vector addresses (PDVAs) from the save file. (See the .POADD function of the PDVOP% monitor call.)

This call can be executed for either sharable or nonsharable save files that were created with the SSAVE% or SAVE% monitor call, respectively. The file must not be open by any process in the user's job. (Refer to the TOPS-20 Monitor Calls Reference Manual for more information regarding the PDVOP%, SSAVE%, and SAVE% monitor calls.)

The GET% monitor call accepts two words of arguments in AC1 and AC2. The first word specifies the handle of the desired process, flag bits, and the JFN of the desired file. The second word specifies where the pages from the file are to be placed in the address space of the process. Thus,

AC1: process handle,,flag bits and a JFN

AC2: lowest process page number in left half, and highest process page number in right half; or the address of an argument block. If this AC contains page numbers, those page numbers control the parts of memory that are loaded when GT%ADR is on in AC1.

Table 5-3 describes the bits that can be set in AC1.

Table 5-3: GET% Flag Bits

Bit	Symbol	Meaning
19	GT%ADR	Use the memory address limits given in AC2. If this bit is off, all existing pages of the file (according to its directory) are mapped.
20	GT%PRL	Preload the pages being mapped (move the pages immediately.) If this bit is off, the pages are read in from the disk when they are referenced.
21	GT%NOV	Do not overlay existing pages and do return an error. If this bit is off, existing pages will be overlaid.
22	GT%ARG	If this bit is on, AC2 contains the address of an argument block.
24-25	GT%JFN	JFN of the save file.

The format of the argument block is described in Table 5-4.

Table 5-4: GET% Argument Block

Word	Symbol	Meaning
0	.GFLAG	Flags that indicate how the rest of the argument block is to be used.
1	.GLOW	Number of the lowest page in the process into which a file page gets loaded. This page must be within the section specified by .GBASE.
2	.GHIGH	Number of the highest page in the process into which a file page gets loaded. This page must be within the section specified by .GBASE.
3	.GBASE	Number of the section into which the file pages are loaded. You can specify the section for single-section save files only; use of this word with a multiple-section save file causes an error. The file pages are loaded into this section of memory regardless of the section specified in the save file.

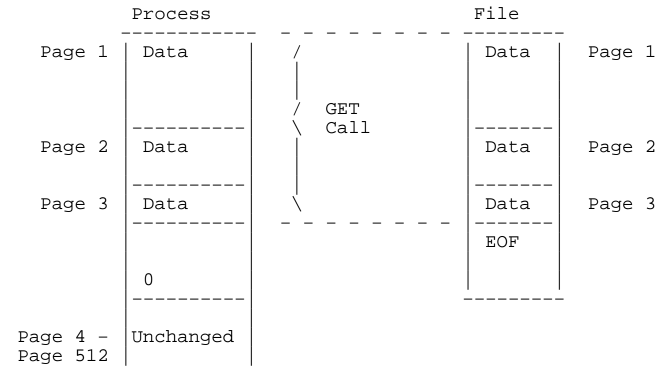
Table 5-5 describes the flag bits defined for use in .GFLAG.

Table 5-5: GET% Argument Block Flags

Bit	Symbol	Meaning
0	GT%LOW	.GLOW contains the number of the lowest page within the process to use.
1	GT%HGH	.GHIGH contains the number of the highest page within the process to use.
2	GT%BAS	.GBASE contains the number of the section to use.

When the pages of the file are mapped into pages in the process's address space, the previous contents of the process pages are overwritten. Any full pages in the process that are not overwritten are unchanged. Any portions of process pages for which there is no data in the file are filled with zeros.

For example, a GET% call executed for a file that contains 2 1/2 pages sets up the process' address space as shown in the following diagram.



After execution of the GET% call, control returns to the user's program at the instruction following the call. If an error occurs, a software interrupt is generated, which the program can process via the software interrupt system.

5.6.2 PMAP% Monitor Call

The PMAP% monitor call is used to map pages from one process to the address space of a second process. Data is not actually transferred; only the contents of the page map of the second (that is, destination) process are changed.

The PMAP% monitor call accepts three words of arguments in AC1 through AC3. The first word contains the handle and page number of the first page to be mapped in the source process (that is, the process whose pages are being mapped). The second word contains the handle and page number of the first page to be mapped in the destination process (that is, the process into which the pages are being mapped). The third

word contains a count of the number of pages to map and bits indicating the access that the destination process will have to the pages mapped. Thus,

AC1: source process handle in the left half, and page number in the process in the right half.

AC2: destination process handle in the left half, and page number in the process in the right half.

AC3: count of number of pages to map and the access bits.

The count and access bits that can be specified in AC3 are described in Section 3.5.6.1.

Upon successful execution of the PMAP% call, addresses in the destination process actually refer to addresses in the source process. The contents of the destination page previous to the execution of the call have been deleted. The access requested in the PMAP% call is granted if it does not conflict with the current access of the destination page (that is, an AND operation is performed between the specified access and the current access). Control returns to the user's program at the instruction following the PMAP% call. If an error occurs, an illegal instruction trap is generated, which the program can process via the software interrupt system or with an ERJMP or ERCAL instruction.

5.7 STARTING AN INFERIOR PROCESS

A program in an inferior process can be started in one of two ways. As mentioned in Section 5.5, the superior process can specify in the CFORK% call the PC for the inferior process and start its execution. Alternatively, the superior process, after executing the CFORK% call to create an inferior process, can execute the SFORK% (Start Process) monitor call to start it.

The SFORK% monitor call accepts two words of arguments in AC1 and AC2.

AC1: flags,,process handle

Flags:

SF%CON(1B0) Used to continue a process that has previously halted. If SF%CON is set, the address in AC2 is ignored, and the process continues from where it was halted.

AC2: the PC of the process being started. The PC contains flags in the left half and the process starting address in the right half. This call obtains the section number of the PC from the entry vector of the process.

There are two alternative ways to start processes: XSFRK% (see Section 8.3.2) or SFRKV% (see the TOPS-20 Monitor Calls Reference Manual).

The process handle given in AC1 cannot refer to a superior process, to more than one process (for example, .FHINF), or to a process that has already been started.

After execution of the SFORK% call, control returns to the user's program at the instruction following the call. If an error occurs, a software interrupt is generated, which the program can process via the software interrupt system.

5.8 INFERIOR PROCESS TERMINATION

The superior process has one of two ways in which it can be notified when one or more of its inferiors terminate execution: via the software interrupt system or by executing the WFORK% monitor call. An inferior process will terminate normally when it executes a HALTF% monitor call. Alternatively, the process will terminate abnormally when it executes an instruction that generates a software interrupt, such as an illegal instruction, and it has not activated the appropriate channel.

By activating channel .ICIFT (channel 19) for inferior process termination and enabling the software interrupt system, the superior process will receive an interrupt when one of its inferiors terminates. (Refer to Section 4.6 for information on activating channel .ICIFT.) The interrupt occurs when any inferior process terminates. Use of the interrupt system allows the superior to do other processing until an interrupt occurs, indicating that an inferior process has terminated.

In some cases, however, the superior cannot do additional processing until either a specific process or all of its inferior processes have completed execution. If this is the case, the superior process can execute the WFORK% (Wait Process) monitor call. This call blocks the superior until one or all of its inferiors have terminated.

The WFORK% monitor call accepts one argument in AC1, the handle of the desired process. This handle can be .FHINF (-4) to block the superior until all inferiors terminate, but cannot be a handle on a superior process.

After execution of the WFORK% monitor call, control returns to the user's program at the instruction following the call, when the specified process or all of the inferior processes terminate. If an error occurs, it generates a software interrupt, which the program can process via the software interrupt system.

5.9 INFERIOR PROCESS STATUS

The superior process can obtain the status of one of its inferiors by executing the RFSTS% (Read Process Status) monitor call. This call returns the status and PC words of the given inferior process.

The short form of the RFSTS% monitor call accepts one argument in AC1, the handle of the desired process. This handle cannot refer to a superior process or to more than one process. The long form accepts two argument words: flags,, process handle in AC1 and the address of the status return block in AC2. In the long form, RF%LNG (bit 0) is set in AC1 and bits 1-17 are unused (must be zero).

After execution of the short form of the RFSTS% call, control returns to the user's program at the instruction following the call. If the RFSTS% call is successful, AC1 contains the status word of the given process and AC2 contains the PC word. The status word is shown in Table 5-6.

Table 5-6: Process Status Word

Bit	Symbol	Meaning		
0	RF%FRZ	The process is suspended (that is, frozen). If this bit is not on, the process is not suspended.		
1-17	RF%STS	The status of the process.		
		Value	Symbol	Meaning
		0	.RFRUN	The process is runnable.
		1	.RFIO	The process is halted waiting for I/O
		2	.RFHLT	The process is halted by a HFORK% or HALTF% monitor call or was never started.
		3	.RFFPT	The process is halted by the occurrence of a software interrupt for which it was not prepared to handle. The right half of the status word contains the number of the channel on which the interrupt occurred.
		4	.RFWAT	The process is halted waiting for another process to terminate.
		5	.RFSLP	The process is halted for a specified amount of time.
		6	.RFTRP	The process is dismissed because it was intercepted by its superior.
		7	.RFABK	The process is dismissed because address break was encountered.
18-35	RF%SIC	The channel number on which an interrupt occurred, which the process was not prepared to handle (see process status code .RFFPT above).		

The RFSTS% call returns with -1 (fullword) in AC3 if the specified handle is assigned but refers to a deleted process. The call generates an illegal instruction interrupt if the handle is unassigned.

In the long form of the RFSTS% monitor call, RF%LNG is set in AC1 and AC2 contains the address of a status-return block. On the return, AC1 and AC2 are not modified. The status-return block is described in Table 5-7.

Table 5-7: RFSTS% Status-Return Block

Word	Symbol	Meaning
0	.RFCNT	Count of words returned in this block in the left half, and count of maximum number of words to return in right half (including

		this word). The right half of this word is specified by the user.
1	.RFPSW	Process status word. This word has the same format as AC1 on a return from a short call. If a valid, but unassigned, process handle was specified in AC1, then this word contains -1 and no other words are returned.
2	.RFPFL	Process PC flags. These are the same flags returned in AC2 on a short call.
3	.RFPPC	Process PC. This is the address; no flags are returned in this word.
4	.RFSFL	Status flag word. Flags: Bit / Symbol / Meaning — B0 / RF%EXO / Process is execute-only.

If an error occurs during execution of the RFSTS% call, a software interrupt is generated which the program can process via the software interrupt system.

5.10 PROCESS COMMUNICATION

A superior process can communicate with its inferiors by sharing the same pages of memory. This sharing is accomplished with the CFORK% (bit CR%MAP) or the PMAP% monitor call. When the superior executes either of these calls, both the superior and the inferior share the same pages. Changes made to the shared pages by either process will be seen by the other process.

Alternatively, processes can communicate via the software interrupt system. The superior process can cause a software interrupt to be generated in an inferior process by executing the IIC% (Initiate Interrupt on Channel) monitor call. For this type of communication to occur, the inferior's interrupt channels must be activated and its interrupt system enabled.

The IIC% monitor call accepts two words of arguments in AC1 and AC2. The handle of the process to receive the interrupt is given in the right half of AC1. AC2 contains a 36-bit word, with each bit representing one of the 36 software channels. If a bit is on in AC2,

a software interrupt is initiated on the corresponding channel. For example, if bit 5 is on in AC2, an interrupt is initiated on channel 5.

Thus,

AC1: process handle in the right half

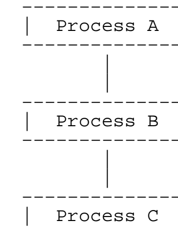
AC2: 36-bit word, with bit n on to initiate a software interrupt on channel n

The process handle given cannot refer to a superior process or to more than one process.

After execution of the IIC% call, control returns to the user's program at the instruction following the call. If an error occurs, it generates a software interrupt which the program can process via the software interrupt system.

5.11 DELETING AN INFERIOR PROCESS

A process is deleted from the job structure when the superior process executes the KFORK% (Kill Process) monitor call. When a process is deleted, its address space, its handle, and any JFNs acquired by the process are released. If the process being deleted has processes inferior to it, the inferiors are also deleted. For example, in the structure:



if process A deletes process B by executing a KFORK% call, process C is also deleted.

The KFORK% monitor call accepts one argument in the right half of AC1, the handle of the process to be deleted. This handle cannot refer to a superior process, to more than one process (for example, .FHINF), or

to the process executing the call (that is, .FHSLF). The RESET% monitor call is used to reinitialize the current process; refer to Section 2.6.1.

After execution of the KFORK% call, control returns to the user's program at the instruction following the call. If an error occurs, a software interrupt is generated, which the program can process via the software interrupt system.

5.12 PROCESS EXAMPLES

Example 1 - This program creates an inferior process to provide timing interrupts.

```
        TITLE TIMINT - AN INFERIOR PROCESS PROVIDING TIMING INTERRUPTS

        SEARCH MONSYM
        SEARCH MACSYM
        .REQUIRE SYS:MACREL

        STDAC.                  ;DEFINE STANDARD ACS

START:  RESET%                  ;RELEASE FILES, ETC.
        MOVE P,[IOWD PDLSIZ,PDL] ;INITIALIZE STACK
        MOVX T1,CR%MAP          ;MAKE NEW PROCESS SHARE THIS
                                ;PROCESS'S MEMORY
        CFORK%                  ;CREATE A NEW PROCESS
         EJSHLT                 ;UNEXPECTED FATAL ERROR
        MOVEM T1,HANDLE         ;SAVE PROCESS HANDLE

;HERE TO START THE INFERIOR PROCESS

STPROC: SETZB T4,FLAG           ;INITIALIZE COUNTER AND FLAG
        MOVE T1,HANDLE          ;GET PROCESS HANDLE
        MOVEI T2,SLEEP          ;GET ADDRESS TO START PROCESS
        SFORK%                  ;START THE NEW PROCESS
         EJSHLT                 ;UNEXPECTED FATAL ERROR

; MAIN PROCESSING LOOP

LOOP:   AOS T4                  ;INCREMENT COUNTER
        SKIPN FLAG              ;HAS TIME ELAPSED YET?
        JRST LOOP               ;NO, GO DO MORE PROCESSING

; HERE WHEN LOWER PROCESS HAS INTERRUPTED

        TMSG <
Counter has reached >           ;OUTPUT FIRST PART OF MESSAGE
        MOVX T1,.PRIOU          ;GET PRIMARY OUTPUT DESIGNATOR
        MOVE T2,T4              ;GET VALUE OF COUNTER
```



```
        MOVEI T3,^D10           ;USE DECIMAL RADIX
        NOUT%                   ;OUTPUT CURRENT COUNTER VALUE
         EJSERR                 ;PRINT ERROR MESSAGE AND CONTINUE
        TMSG <
>                               ;MOVE TO A NEW LINE
        JRST STPROC             ;CONTINUE COUNTING

; PROGRAM PERFORMED BY INFERIOR PROCESS TO WAIT FOR ONE-HALF MINUTE

SLEEP:  MOVX T1,^D30*^D1000     ;ONE-HALF MINUTE IN MILLISECONDS
        DISMS%                  ;WAIT FOR SPECIFIED TIME
        SETOM FLAG              ;TELL SUPERIOR TIME HAS ELAPSED
        HALTF%                  ;FINISHED

; CONSTANTS AND STORAGE

        PDLSIZ==50              ;SIZE OF THE STACK
PDL:    BLOCK PDLSIZ            ;STACK
HANDLE: BLOCK 1                 ;INFERIOR PROCESS HANDLE
FLAG:   BLOCK 1                 ;INTERRUPT FLAG

        END START
```

Example 2 - This program illustrates how an inferior process may be used as a source of timer interrupts. The main program increments a counter. It has an inferior process running for the sole purpose of timing 10 second intervals. Each time the inferior process has timed 10 seconds, it stops and interrupts the main program. The main program then reports how many more times it has incremented the counter since the last 10 second interrupt.

```
        TITLE TRMINT - FORK TERMINATION INTERRUPTS
        SEARCH MONSYM
        SEARCH MACSYM
        .REQUIRE SYS:MACREL

        STDAC.                  ;DEFINE STANDARD ACS

START:  RESET%                  ;RELEASE FILES, ETC.
        MOVE P,[IOWD PDLSIZ,PDL] ;INITIALIZE STACK

; SET UP THE INTERRUPT SYSTEM

        MOVX T1,.FHSLF          ;GET PROCESS HANDLE FOR THIS FORK
        MOVE T2,[LEVTAB,,CHNTAB] ;GET TABLE ADDRESSES
        SIR%                    ;SET INTERRUPT TABLE ADDRESSES
         EJSHLT                 ;UNEXPECTED FATAL ERROR
        MOVX T2,1B<.ICIFT>      ;GET PROCESS TERMINATION CHANNEL BIT
        AIC%                    ;ACTIVATE PROCESS TERMINATION CHANNEL
         EJSHLT                 ;UNEXPECTED FATAL ERROR
        EIR%                    ;ENABLE INTERRUPT SYSTEM
         EJSHLT                 ;UNEXPECTED FATAL ERROR
```

```
; CREATE AND START THE INFERIOR PROCESS

        MOVX T1,CR%MAP+CR%ST+SLEEP
        CFORK%                  ;CREATE AND START TIMER AT SLEEP
         EJSHLT                 ;UNEXPECTED FATAL ERROR
        MOVEM T1,HANDLE ;SAVE PROCESS HANDLE

;INITIALIZE THE COUNTER

STPROC: SETZB T4,OLDT4          ;CLEAR COUNTER

;MAIN LOOP OF THE PROGRAM WHICH JUST KEEPS COUNTING. (REAL
;APPLICATION WOULD PRESUMABLY HAVE A MORE USEFUL MAIN PROGRAM.)

LOOP:   AOJA T4,LOOP            ;JUST KEEP INCREMENTING
; HERE WHEN LOWER PROCESS HAS INTERRUPTED

PROINT: MOVEM P,IACS+P          ;SAVE STACK POINTER
        MOVEI P,IACS            ;MAKE POINTER FOR REST OF ACS
        BLT P,IACS+CX           ;SAVE REST OF ACS
        MOVE P,IACS+P           ;RESTORE P
        TMSG <NUMBER OF COUNTS: >
        MOVX T1,.PRIOU          ;GET PRIMARY OUTPUT DESIGNATOR
        EXCH T4,OLDT4           ;SAVE NEW COUNTER VALUE
        SUB T4,OLDT4            ;FIND NUMBER OF COUNTS SINCE LAST TIME
        MOVM T2,T4              ;MAKE IT POSITIVE
        MOVEI T3,^D10           ;USE DECIMAL RADIX
        NOUT%                   ;OUTPUT CURRENT COUNTER VALUE
         EJSERR                 ;PRINT ERROR MESSAGE AND CONTINUE
        TMSG <
>                               ;MOVE TO A NEW LINE
        MOVE T1,HANDLE          ;GET PROCESS HANDLE
        MOVEI T2,SLEEP          ;GET ADDRESS TO START PROCESS
        SFORK%                  ;START THE NEW PROCESS
         EJSHLT                 ;UNEXPECTED FATAL ERROR
        MOVSI P,IACS            ;GET POINTER TO SAVED ACS
        BLT P,P                 ;RESTORE SAVED ACS
        DEBRK%                  ;DISMISS INTERRUPT
;THE FOLLOWING IS EXECUTED AS A LOWER PROCESS TO DO THE
;TIMING.  IT SLEEPS FOR 10 SECONDS AND THEN STOPS.

SLEEP:  MOVX T1,^D10*^D1000     ;10 SECONDS IN MILLISECONDS
        DISMS%                  ;SLEEP
        HALTF%                  ;STOP AND INTERRUPT THE MAIN PROGRAM

; CONSTANTS AND STORAGE

        PDLSIZ==50              ;SIZE OF THE STACK
PDL:    BLOCK PDLSIZ            ;STACK
CHNTAB: REPEAT ^D19,<EXP 0>     ;CHANNELS 0-18 ARE NOT USED
        1,,PROINT               ;LEVEL 1 PROCESS TERMINATION CHANNEL
        REPEAT ^D15,<EXP 0>     ;REMAINING CHANNELS ARE NOT USED
```



```
LEVTAB: RETPC1                  ;RETURN PC STORED AT RETPC1 FOR
                                ;LEVEL 1
        0                       ;LEVEL 2 NOT USED
        0                       ;LEVEL 3 NOT USED
HANDLE: BLOCK 1                 ;INFERIOR PROCESS HANDLE
RETPC1: BLOCK 1                 ;RETURN PC STORED HERE ON INTERRUPTS
OLDT4:  BLOCK 1                 ;HOLDS TIMER VALUE AT LAST INTERRUPT
IACS:   BLOCK 20                ;STORAGE FOR ACS DURING INTERRUPTS

        END START
```

Example 3 - This program creates an inferior process which waits until a line has been typed on the terminal.

```
        TITLE FRKDOC - AN INFERIOR PROCESS WAITS UNTIL A LINE IS TYPED

        SEARCH MONSYM
        SEARCH MACSYM
        .REQUIRE SYS:MACREL

        STDAC.                  ;DEFINE STANDARD ACS

START:  RESET%                  ;RELEASE FILES, ETC.
        MOVE P,[IOWD PDLSIZ,PDL] ;INITIALIZE STACK
        MOVX T1,CR%MAP          ;MAKE NEW PROCESS SHARE THIS
                                ;PROCESS'S MEMORY
        CFORK%                  ;CREATE A NEW PROCESS
         EJSHLT                 ;UNEXPECTED FATAL ERROR
        SETZB T4,FLAG           ;INITIALIZE COUNTER AND FLAG
        MOVEI T2,GETCOM         ;GET ADDRESS TO START PROCESS
        SFORK%                  ;START THE NEW PROCESS
         EJSHLT                 ;UNEXPECTED FATAL ERROR

; MAIN PROCESSING LOOP

LOOP:   AOS T4                  ;INCREMENT COUNTER
        SKIPN FLAG              ;HAS TIME ELAPSED YET?
        JRST LOOP               ;NO, GO DO MORE PROCESSING

; HERE WHEN INFERIOR PROCESS HAS INPUT A LINE OF TEXT

        TMSG <
Counter has reached >           ;OUTPUT FIRST PART OF MESSAGE
        MOVX T1,.PRIOU          ;GET PRIMARY OUTPUT DESIGNATOR
        MOVE T2,T4              ;GET VALUE OF COUNTER
        MOVEI T3,^D10           ;USE DECIMAL RADIX
        NOUT%                   ;OUTPUT CURRENT COUNTER VALUE
         EJSERR                 ;PRINT ERROR MESSAGE AND CONTINUE
        TMSG <
Echo Check: >                   ;OUTPUT FIRST PART OF MESSAGE
        HRROI T1,BUFFER         ;GET POINTER TO BUFFER
        PSOUT%                  ;OUTPUT TEXT JUST ENTERED
```

```
        HALTF%                  ;STOP
        JRST START              ;IN CASE PROGRAM CONTINUED

; PROGRAM PERFORMED BY INFERIOR PROCESS TO INPUT A LINE OF TEXT

GETCOM: HRROI T1,BUFFER         ;GET POINTER TO TEXT BUFFER
        MOVEI T2,BUFSIZ*5       ;GET COUNT OF MAX # OF CHARACTERS
        SETZM T3                ;NO RETYPE BUFFER
        RDTTY%                  ;READ A LINE FROM THE TERMINAL
         EJSERR                 ;UNEXPECTED ERROR
        SETOM FLAG              ;TELL SUPERIOR TIME HAS ELAPSED
        HALTF%                  ;FINISHED

; CONSTANTS AND STORAGE

        PDLSIZ==50              ;SIZE OF THE STACK
PDL:    BLOCK PDLSIZ            ;STACK
        BUFSIZ==50              ;BUFFER SIZE
BUFFER: BLOCK BUFSIZ
FLAG:   BLOCK 1                 ;INTERRUPT FLAG

        END START
```

CHAPTER 6

ENQUEUE/DEQUEUE FACILITY

6.1 OVERVIEW

Many times users are placed in situations where they must share files with other users. Each user wants to be guaranteed that while reading a file, other users are reading the same data and while writing a file, no users are also writing, or even reading, the same portion of the file.

Consider a data file used by members of an insurance company. When many agents are reading individual accounts from the data file, they can all access the file simultaneously because no one is changing any portion of the data. However, when an agent desires to modify or replace an individual account, that portion of the file should be accessed exclusively by that agent. None of the other agents wants to access accounts that are being changed until after the changes are made.

By using the ENQ/DEQ facility, cooperating users can insure that resources are shared correctly and that one user's modifications do not interfere with another user's. Examples of resources that can be controlled by this facility are devices, files, operations on files (for example, READ, WRITE), records, and memory pages. This facility can be used for dynamic resource allocation, computer networks, and internal monitor queueing. However, control of simultaneous updating of files by multiple users is its most common application.

The ENQ/DEQ facility insures data integrity among processes only when the processes cooperate in their use of both the facility and the physical resource. Use of the facility does not prevent non-cooperating processes from accessing a resource without first enqueueing it. Nor does the facility provide protection from processes using it in an incorrect manner.

A resource is defined by the processes using it and not by the system. Because there is competition among processes for use of a resource, each resource is associated with a queue. This queue is the ordering of the requests for the resource. When a request for the resource is

granted, a lock occurs between the process that made the request and the resource. For the duration of the lock, that process is the owner of the resource. Other processes requesting access to the resource are placed in the queue until the owner relinquishes the lock. However, there can be more than one owner of a resource at a time; this is called shared ownership (refer to Section 6.2). Processes obtain access to a specific resource by placing a request in the queue for the resource. This request is generated by the ENQ% monitor call. When finished with the resource, the process then issues the DEQ% monitor call. This call releases the lock by removing the request from the queue and makes the resource available to the next waiting process. This cycle continues until all requests in the queue have been satisfied.

6.2 RESOURCE OWNERSHIP

Ownership for a resource can be requested as either exclusive or shared. Exclusive ownership occurs when a process requests sole use of the resource. When a process is granted exclusive ownership, no other process will be allowed to use the resource until the owner relinquishes it. This type of ownership should be requested if the process plans on modifying the resource (for example, the process is updating a record in a data file). Shared ownership occurs when a process requests a resource, specifying that it will share the use of the resource with other processes. When a process is given shared ownership, other processes also specifying shared ownership are allowed to simultaneously use the resource. Access to a resource should be shared as long as any one process is not modifying the resource.

Two conditions determine when a lock to a resource is given to a process:

1. The position of the process's request in the queue for the resource.

2. The type of ownership specified by the process's request.

Because each resource has only one queue associated with it, requests for both exclusive and shared ownership of the resource are placed in the same queue. Requests are placed in the queue in the order in which the ENQ facility receives them, and the first request in the queue will be the first one serviced (except in the case of single requests for multiple resources; refer to Section 6.4.1). In other words, the ENQ facility processes requests on a first in, first out basis. If this first request is for shared ownership, that request will be serviced along with all following shared ownership requests up to but not including the first exclusive ownership request. If the first request is for exclusive ownership, no other processes are allowed use of the resource until the first process has released the lock.

Consider the following queue for a particular resource.

```
!======================================================!
!                 request 1 (shared)                    !
!-------------------------------------------------------!
!                 request 2 (shared)                    !
!-------------------------------------------------------!
!                request 3 (exclusive)                  !
!-------------------------------------------------------!
!                 request 4 (shared)                    !
!-------------------------------------------------------!
!                 request 5 (shared)                    !
!======================================================!
```

Request 1 will be serviced first because it is the first request in the queue. However, since this request is for shared ownership, request 2 can also be serviced. Request 3 cannot be serviced until the processes with request 1 and request 2 release the lock on the resource. Eventually the lock is released by the two processes, and the first two requests are removed from the queue. The queue now has the following entries:

```
!======================================================!
!                                                       !
!-------------------------------------------------------!
!                                                       !
!-------------------------------------------------------!
!                request 3 (exclusive)                  !
!-------------------------------------------------------!
!                 request 4 (shared)                    !
!-------------------------------------------------------!
!                 request 4 (shared)                    !
!======================================================!
```

Request 3 is now first in the queue and is given a lock on the resource. Because the request is for exclusive ownership, no other requests will be serviced. Once the process associated with request 3 releases the lock, both request 4 and request 5 can be serviced because they both are for shared ownership.

6.3 PREPARING FOR THE ENQ/DEQ FACILITY

Before using the ENQ/DEQ facility, the user must obtain an ENQ quota from the system administrator and must obtain the name of the resource

desired, the type of protection required, and the level number associated with the resource.

The ENQ quota indicates the total number of requests that can be outstanding for the user at any given time. Any request that would cause the quota to be exceeded results in an error. The user cannot use the ENQ facility if the quota is set to zero.

The resource name has a meaning agreed upon by all users of the specific resource and serves as an identifier of the resource. The system makes no association between the resource name and the physical resource itself; it is the responsibility of the user's process to make that association. The system merely uses the resource name to process requests and handles different resource names as requests for different resources.

The resource name has two parts. In most cases, the first part is the JFN of the file being accessed. Before using the ENQ facility, the user must initialize the file using the appropriate monitor calls (refer to Section 3.1). The second part of the name is a modifier, which is either a pointer to a string or a 33-bit user code. The string uniquely identifies the resource to all users. The pointer can either be a standard byte pointer or be in the form

```
-1,,ADR
```

where ADR is the location of the left-justified ASCIZ text string. The 33-bit user code similarly identifies the resource by representing an item such as a record number or block number. The ENQ facility considers these modifiers as logical strings and does not check for cooperation among the users. Thus, users must be careful when assigning these modifiers to prevent the occurrence of two different modifiers referring to the same resource.

The type of protection desired for the resource is indicated by the first part of the resource name. This part of the name can be one of four values. When the user specifies the JFN of the desired file, the file is subject to the standard access protection of the system. This is the most typical case. When the user specifies -1 instead of a JFN, it means that resources defined within a job are to be accessed only by processes of that job. Other jobs requesting resources of the same name are queued to a different resource. When the user specifies -2 instead of a JFN, it means that the resource can be accessed by any job on the system. A process must have bit SC%ENQ enabled in its capability word to specify this type of protection. If the user specifies -3 instead of a JFN, it means the same type of protection as that given when -2 is specified. However, this requires that the process have WHEEL or OPERATOR capability enabled. Quotas are not checked when -3 is given instead of a JFN.

In addition to specifying the resource name and type of protection, the user also assigns a level number to each resource. The use of

level numbers prevents the occurrence of a deadly embrace situation: the situation where two or more processes are waiting for each to complete, but none of the processes can obtain a lock on the resource it needs for completion. This situation is represented by Figure 6-1.

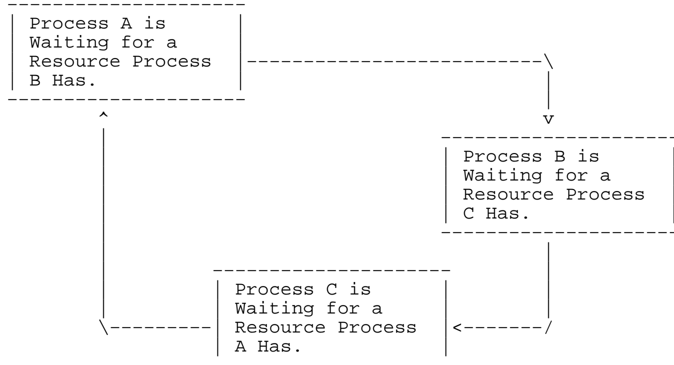


Figure 6-1: Deadly Embrace Situation

Each process is in the queue waiting for the resource it needs, but no request is being serviced because the desired resources are unavailable.

The use of level numbers forces cooperating processes to order their use of resources by requiring that processes request resources in an ascending numerical order and that all processes assign the same level number to a specific resource. This means that the order in which resources are requested is the same for all processes and therefore, requests for the first resource will always precede requests for the second one.

If both of the above requirements are not met, the process requesting the resource receives an error, unless the appropriate flag bit is set (refer to Section 6.4.1.2), and the request is not placed in the queue. Thus, instead of waiting for a resource it will never get, the process is informed immediately that the resource is not available.

ENQUEUE/DEQUEUE FACILITY

6.4 USING THE ENQ/DEQ FACILITY

There are three monitor calls available for the ENQ/DEQ facility: ENQ%, to request use of a resource; DEQ%, to release a lock on a resource; and ENQC%, to obtain information about the queues and to specify access to these queues.

6.4.1 Requesting Use of a Resource

The user issues the ENQ% monitor call to place a request in the queue associated with the desired resource. This call is used to specify the resource name, level number, and type of protection required.

A single ENQ% monitor call can be used to request any number of resources. In fact, when desiring multiple resources, the user should request all of them in one call. This method of requesting resources guarantees that the user gets either none or all of the resources requested because the ENQ/DEQ facility never allocates only some of the resources specified in one call. Because all resources in a single call must be available at the same time, the first user requesting a resource (that is, the first user in the queue for the resource) may not be the first user obtaining it if other resources in the request are currently not available.

A single call for multiple resources is not functionally the same as a series of single calls of those resources. In a single call, the entire request is rejected if an error is returned for one of the resources specified. In a series of single calls, each request that did not return an error will be queued.

The ENQ% monitor call accepts two words of arguments in AC1 and AC2. The first word contains the code of the desired function, and the second contains the address of the argument block. Thus,

AC1: function code

AC2: address of argument block

6.4.1.1 ENQ% Functions - The functions that can be requested in the ENQ% call are described in Table 6-1.

Table 6-1: ENQ% Functions

Code	Symbol	Meaning
0	.ENQBL	Queue the requests and block the process until all requested locks are acquired. This function returns an error code only if the ENQ% call is not correctly specified.
1	.ENQAA	Queue the requests and acquire the locks only if all requested resources are immediately available. If the resources are available, all will be allocated to the process. If any one of the resources is not available, no requests are queued, no locks are acquired, and an error code is returned in AC1.
2	.ENQSI	Queue the requests for all specified resources. If all resources are available, this function is identical to the .ENQBL function. If all resources are not immediately available, the requests are queued, and a software interrupt is generated when all requested resources have been given to the process.
3	.ENQMA	Change the ownership access of a previously-queued request (refer to bit EN%SHR below). The access for each lock in this request is compared with the access for each lock in the request already queued. No action is taken if the two accesses are the same. If the access in this request is shared and the access in the previous request is exclusive, the ownership access is changed to shared access. Otherwise, an error is returned if: 1. There are processes which are locking, or waiting on the same lock, and the process tries to change the ownership access from shared to exclusive. If this is the case, the process should issue a DEQ% monitor call for the shared request and then issue another ENQ% monitor call for exclusive ownership. 2. Any one of the specified locks does not have a pending request. 3. Any one of the specified locks is a pooled resource (refer to Section 6.4.1.2). Each lock specified is checked, and the access is changed for all locks that were correctly given. On receiving an error, the process should issue the ENQC% monitor call to determine the current state of each lock (refer to Section 6.4.3).
4	.ENECL	Set cluster-wide ENQ/DEQ functionality for all ENQ/DEQ/ENQC JSYSes performed by this process. The contents of AC2 are ignored as this function does not require an argument block.

ENQUEUE/DEQUEUE FACILITY

6.4.1.2 ENQ% Argument Block - The format of the argument block is described in Table 6-2.

Table 6-2: ENQ% Argument Block

Word	Symbol	Meaning
0	.ENQLN	Number of locks being requested in the left half, and length of argument block (including this word) in the right half.
1	.ENQID	Number of software interrupt channel in the left half, and request ID in the right half.
2	.ENQLV	Flags and level number in the left half, and JFN, -1, -2 or -3 (refer to Section 6.3) in the right half.

3	.ENQUC	Pointer to string or 5B2+33-bit user code (refer to Section 6.3).
4	.ENQRS	Number of resources in the pool in the left half, and number of resources requested in the right half.
5	.ENQMS	Address of a resource mask block.

Words .ENQLV, .ENQUC, .ENQRS, and .ENQMS (words 2 through 5) are repeated for each lock being requested. These four words are called the lock specification.

Software Interrupts

The software interrupt system is used in conjunction with the .ENQSI function (refer to Section 6.4.1.1). If all locks are not available when the user requests them, the .ENQSI function causes a software interrupt to be generated when the locks become available. The user specifies the software channel on which to receive the interrupt by placing the channel number in the left half of word .ENQID in the argument block.

When the user is waiting for more than one lock to become available, he will receive an interrupt when the last lock is available. If he desires to be informed as each lock becomes available, he can assign the locks to separate channels by issuing multiple ENQ% calls. The availability of each lock will then be indicated by the occurrence of an interrupt on each channel.

When the user requests the .ENQSI function, he must initialize the interrupt system first or else an interrupt will not be generated when the locks become available (refer to Chapter 4).

Request ID

The 18-bit request ID is currently not used by the system, but is stored for use by the process. Thus, the process can supply an ID to use as identification for the request. This ID is useful on the .DEQID function of the DEQ% monitor call (refer to Section 6.4.2.1).

Flags and Level Numbers

Table 6-3 describes the flags that can be used in the left half of the first word of each lock specification (.ENQLV).

Table 6-3: Lock Specification Flags

Bit	Symbol	Meaning
0	EN%SHR	Ownership for this resource is to be shared. If this bit is not on, ownership for this resource is to be exclusive.
1	EN%BLN	Ignore the level number associated with this resource. If this bit is set, sequencing errors in level numbers are not considered fatal, and execution of the call continues. On successful completion of the call, AC1 contains either an error code if a sequencing error occurred, or zero if a sequencing error did not occur. WARNING A deadly embrace situation may occur when level numbers are not used. Use of these numbers guarantees that such a situation cannot arise; for this reason bit EN%BLN should not be set.
2	EN%NST	Allow ownership of this lock to be nested.
3	EN%LTL	Allow a long-term lock on this resource.
4-8		Reserved for Digital.
9-17	EN%LVL	Level number associated with this resource. This number is specified by the user and must be agreed upon by all users of the resource. In order to eliminate a deadly embrace situation, users must request resources in numerically increasing order.

The request is not queued, and an error is given, if EN%BLN is not set and

1. The user requests a resource with a level number less than or equal to the highest numbered resource he has requested so far.

2. The level number of this request does not match the level number supplied in previous requests for this resource.

Pooled Resources

Word .ENQRS of each lock specification is used to allocate multiple copies from a pool of identical resources. Bit EN%SHR, indicating shared ownership, is meaningless for pooled resources because each resource in the pool can be owned by only one process at a time. A process can own one or more resources in the pool; however, it cannot own more than there are in the pool or more than there are unowned in the pool.

The left half of word .ENQRS contains the total number of resources existing in the pool. This number is previously agreed upon by all users of the pooled resource. The first user who requests the resource sets this number, and all subsequent requests must specify the same number or an error is given.

The right half of word .ENQRS contains the number of resources being requested by this process. This number must be greater than zero if a pool of resources exists and cannot be greater than the number in the left half. This means that if a pool of resources exists, the user must request at least one resource, but cannot request more than are in the pool.

Once the number of pooled resources is determined, the resources are allocated until the pool is depleted or until a request specifies more resources than are currently available. In the latter case, the user making the request is not given any resources until his entire request can be satisfied. Subsequent requests from other users are not granted until this request is satisfied even though there may be enough resources to satisfy these subsequent requests. As users release their resources, the resources are returned to the pool. When all resources have been returned, they cease to exist, and the next request completely redefines the number of resources in the new pool.

The system assumes that the resource is in a pool if the left half of word .ENQRS of the lock specification is nonzero. Thus the user should set the left half to zero if only one resource of a specific type exists. If this is the case, then the right half of this word is a number defining the group of users who can simultaneously share the resource. This means that when the resource is allocated to a user for shared ownership, only other users in the same group will be allowed access to the resource. The use of sharer groups restricts

ENQUEUE/DEQUEUE FACILITY

access to a resource to a set of processes smaller than the set for shared ownership (which is sharer group 0) but larger than the set for exclusive ownership. (Refer to Section 6.5 for more information on sharer groups).

6.4.2 Releasing a Resource

The user issues the DEQ% monitor call to remove a request from the queue associated with a resource. The request is removed whether or not the user actually owns a lock on the resource or is only waiting in the queue for the resource.

The DEQ% monitor call can be used to remove any number of requests from the queues. If one of the requests cannot be removed, the dequeueing procedure continues until all lock specifications have been processed. An error code is then returned for the last request found that could not be dequeued. The process can then execute the ENQC% call (refer to Section 6.4.3) to determine the status of each lock. Thus, unlike the operation of the ENQ% call, the DEQ% call will dequeue as many resources as it can, even if an error is returned for one of the lock specifications in the argument block. However, when a user attempts to dequeue more pooled resources than he originally allocated, an error code is returned and none of the resources are dequeued.

The DEQ% monitor call accepts two words of arguments in AC1 and AC2. The first word contains the code for the desired function, and the second word contains the address of the argument block. Thus,

AC1: function code

AC2: address of argument block

6.4.2.1 DEQ% Functions - The DEQ% functions are described in Table 6-4.

Table 6-4: DEQ% Functions

Code	Symbol	Meaning
0	.DEQDR	Remove the specified requests from the queues. This function is the only one that requires an argument block.
1	.DEQDA	Remove all requests for this process from the queues. This action is taken on a RESET monitor call. An error code is returned if this process has not requested any resources (that is, if this process has not issued an ENQ%).
2	.DEQID	Remove all requests that correspond to the specified request identifier. When this function is specified, the user must place the 18-bit request ID in AC2 on the DEQ% call. This function allows the user to release a class of locks in one call without itemizing each lock in an argument block. The function should be used when dequeueing in one call the same locks that were enqueued in one call. For example, with this function the user can specify the ID to be the same as the JFN used in the ENQ% call and thus remove all locks to that file at once.

6.4.2.2 DEQ% Argument Block - The format of the argument block for function .DEQDR is described in Table 6-5.

Table 6-5: DEQ% Argument Block

Word	Symbol	Meaning
0	.ENQLN	Number of locks being requested in the left half, and length of argument block (including this word) in the right half.
1	.ENQID	Number of software interrupt channel in the left half, and request ID in the right half.
2	.ENQLV	Flags and level number in the left half, and JFN, -1, -2 or -3 (refer to Section 6.3) in the right half.
3	.ENQUC	Pointer to string or 5B2+33-bit user code (refer to Section 6.3).
4	.ENQRS	Number of resources in the pool in the left half, and number of resources requested in the right half.
5	.ENQMS	Address of a resource mask block.

Words .ENQLV, .ENQUC, .ENQRS, and .ENQMS (words 2 through 5) are repeated for each request being dequeued. These four words are called the lock specification.

6.4.3 Obtaining Information About Resources

The user issues the ENQC% monitor call to obtain information about the current status of the given resources. This call can also be used by privileged users to perform various utility functions on the queue structure. The format of the ENQC% call is different for these two uses. (Refer to the TOPS-20 Monitor Calls Reference Manual for the explanation of the privileged use of the ENQC% call.)

The ENQC% monitor call accepts three words of arguments in AC1 through AC3:

AC1: function code (.ENQCS)

AC2: address of argument block

AC3: address of area to receive status information

The format of the argument block is identical to the format of the ENQ% and DEQ% argument blocks. The area in which the status is to be returned should be three times as long as the number of locks specified in the argument block.

On successful execution of the ENQC% call, the current status of each lock specified is returned as a three-word entry. This three-word entry has the following format.

```
!=======================================================!
!       Flag bits indicating status of lock             !
!-------------------------------------------------------!
!                  36-bit time stamp                    !
!-------------------------------------------------------!
!        Reserved          !        Request ID          !
!=======================================================!
```

Table 6-6 describes the flag bits that can be used in a ENQC% call.

Table 6-6: ENQC% Flag Bits

Bit	Symbol	Meaning
0	EN%QCE	An error has occurred in the corresponding lock request. Bits 18-35 contain the appropriate error code.
1	EN%QCO	The process issuing the ENQC% call is the owner of this lock.
2	EN%QCQ	The process issuing the ENQC% call is in the queue waiting for this resource. This bit will be on when EN%QCO is on because a request remains in the queue until a DEQ% call is given.
3	EN%QCX	The lock has been allocated for exclusive ownership. When this bit is off, there is no way of determining the number of sharers of the resource.
4	EN%QCB	The process issuing the ENQC% call is in the queue waiting for exclusive ownership to the resource. This bit will be off if EN%QCQ is off.
5	EN%QCC	This is a cluster-wide lock/request. This bit exists in both a lock-block and a q-block.
6	EN%QCN	No future vote is required for this lock. This bit exists in a lock-block.
7	EN%QCS	This lock requires a scheduling pass.
8		Reserved for Digital
9-17	EN%LVL	The level number of the resource.
18-35	EN%JOB	The number of the job that owns the lock. This value may be a job number on another system within the cluster. For locks with shared ownership, this value will be the job number of one of the owners. However, this value will be the current job's number if the current job is one of the owners. If this lock is not owned, the value is -1. If EN%QCE is on, this field contains the appropriate error code.

The 36-bit time stamp indicates the last time a process locked the resource. The time is in the universal date-time standard. If no one currently has a lock on the resource, this word is zero.

The request ID returned in the right half of the third word is either the request ID of the current process if that process is in the queue or the request ID of the owner of the lock.

6.5 SHARER GROUPS

Processes can specify the sharing of resources by using sharer group numbers (refer to Section 6.4.1.2). The use of sharer groups restricts the ownership for a resource to a set of processes smaller than the set for shared ownership but larger than the set for exclusive ownership.

Sharer group number 0 is used to indicate the group of all cooperating processes of the resource. This group number is assumed when no group is specified in the ENQ% call. To restrict use of the resource, a group number other than 0 must be explicitly specified in the call.

Consider the following example. The resource is the WRITE operation on a file. There are four types of uses of this resource as shown in Figure 6-2.

ENQUEUE/DEQUEUE FACILITY

Process' Own Use of the Resource \ Other Process' Use of Resource	Write	Not Allowed to Write
Write	1 Shared, Group 0	2 No Need to Use ENQ/DEQ
Not Allowed to Write	3 Exclusive	4 Shared, Group 1

Figure 6-2: Use of Sharer Groups

In block 1 of the figure, the process owning the lock wishes to allow all cooperating processes to also lock the resource (that is, to perform the WRITE operation). Therefore, in the ENQ% call, the process specifies the resource can be locked by all cooperating processes. In block 2 of the figure, the process does not plan on locking the resource and does not care if other processes lock it. Thus, there is no need for the process to use the ENQ/DEQ facility. In block 3 of the figure, the process desires to lock the resource exclusively and does not want other processes to lock it. Thus, the process obtains exclusive ownership for the resource. In block 4 of the figure, the process does not want to lock the resource immediately but also does not want other processes to lock it because it soon plans to request a lock on the resource. If the process were the only one requesting this type of use, exclusive ownership would be sufficient, because the resource would be unavailable to others as long as the process owned the lock. However, if other processes desire this same type of use, exclusive ownership is not sufficient, because once one process releases the lock, another process with a different type of use could obtain its own lock. Thus, in this example, sharer group 1 is defined to include all processes with the same type of use (that is, all processes who do not want to lock the resource immediately but also do not want other processes to lock it).

This elimates the problem of another user obtaining the resource for a different type of use.

Sharer group 0 should be sufficient for most uses of the ENQ/DEQ facility. Additional groups should only be needed in those situations where a subset of the cooperating processes must have a specific use of a resource, as in the above example.

6.6 AVOIDING DEADLY EMBRACES

Processes can interact in many undesirable ways if improper communication occurs among the processes or if resources are incorrectly shared. An example of one undesirable situation is the occurrence of a deadly embrace: when two processes are waiting for each other to complete but neither one can gain access to the resource it needs for completion. This situation can be avoided when processes consider the following guidelines.

1. Processes should request resources at the time they need them. If possible, processes should request resources one at a time and release each resource before requesting the next one.

2. Processes should request shared ownership whenever possible. However, the process should not request shared ownership if it plans on modifying the resource.

3. When a process needs more than one resource, it should request these resources in one ENQ% call instead of multiple calls for each resource. The process should also release the entire set of resources at once with a single DEQ% call.

4. When the use of one resource depends on the use of a second one, the process should define the two resources as one in the ENQ% and DEQ% calls. However, there is no protection of the resources if they are also requested separately.

5. Occasionally processes use a set of resources and require a lock on the second resource while retaining the lock on the first. In this case, the order in which the locks are obtained should be the same for all users of the set of resources. The same ordering of locks is accomplished by the processes assigning level numbers to each resource. The requirements that processes request resources in ascending numerical order and that all processes use the same level number for a specific resource ensure that a deadly embrace situation will not occur.

CHAPTER 7

INTER-PROCESS COMMUNICATION FACILITY

7.1 OVERVIEW

The Inter-Process Communication Facility (IPCF) allows communication among jobs and system processes. This communication occurs when processes send and receive information in the form of packets. Each sender and receiver has a Process ID (PID) assigned to it for identification purposes.

When the sender sends a packet of information to another process, the packet is placed into the receiver's input queue. The packet remains in the queue until the receiver checks the queue and retrieves the packet. Instead of periodically checking its input queue, the receiver can enable the software interrupt system (refer to Chapter 4) to generate an interrupt when a packet is placed in its input queue.

The <SYSTEM>INFO process is the information center for the Inter-Process Communication Facility. This process performs system functions related to PIDs and names, and any process can request these functions by sending <SYSTEM>INFO a packet.

7.2 QUOTAS

Before using IPCF, the user must acquire the ability to use IPCF privileges from the system administrator: a send packet quota and a receive packet quota. These quotas designate, on a per process basis, the number of sends and receives that can be outstanding at any one time. For example, if the process has a send quota of two and it has sent two packets, it cannot send any more until at least one packet has been retrieved by its receiver. A send packet quota of two and a receive packet quota of five are assumed as the standard quotas. If these quotas are zero, the process cannot use IPCF.

7.3 PACKETS

Information is transferred in the form of packets. Each packet is divided into two portions: a packet descriptor block of four to six words and a packet data block the length of the message. The format of the packet is shown in Figure 7-1.

```
Packet Descriptor Block

        !=====================================================!
.IPCFL  !                        flags                        !
        !-----------------------------------------------------!
.IPCFS  !                    PID of sender                    !
        !-----------------------------------------------------!
.IPCFR  !                   PID of receiver                   !
        !-----------------------------------------------------!
.IPCFP  !    length of message     !    address of message    !
        !            n             !           ADR            !
        !-----------------------------------------------------!
.IPCFD  !   sender's connected     !   sender's logged in     !
        !        directory         !        directory         !
        !-----------------------------------------------------!
.IPCFC  !           enabled capabilities of sender            !
        !-----------------------------------------------------!
.IPCSD  !            connected directory of sender            !
        !-----------------------------------------------------!
.IPCAS  !               account string of sender              !
        !-----------------------------------------------------!
.IPCLL  !              logical location of sender             !
        !=====================================================!


Packet Data Block

        !=====================================================!
ADR     !                    message word 1                   !
        !=====================================================!
                                  .
                                  .
                                  .
        !=====================================================!
        !                    message word n                   !
        !=====================================================!
```

Figure 7-1: IPCF Packet

7.3.1 Flags

There are two types of flags that can be set in word .IPCFL of the packet descriptor block. The flags in the left half of the word are instructions to IPCF for packet communication, and the flags in the right half are descriptions of the data message. The flags in the right half are returned as part of the associated variable (refer to Section 7.4.2). The packet descriptor block flags are described in Table 7-1.

Table 7-1: Packet Descriptor Block Flags

Bit	Symbol	Meaning
0	IP%CFB	Do not block the process if there are no messages in the queue. If this bit is on, the process receives an error if there are no messages.
1	IP%CFS	Use the PID obtained from the address in word .IPCFS of the packet descriptor block as the sender's PID.
2	IP%CFR	Use the PID obtained from the address in word .IPCFR of the packet descriptor block as the receiver's PID.
3	IP%CFO	Allow the process one send above the send quota. (The standard send quota is two.)
4	IP%TTL	Truncate the message if it is longer than the area reserved for it in the packet data block. If this bit is not on, the process receives an error if the message is too long.
5	IP%CPD	Create a PID to use as the sender's PID. The PID created is returned in word .IPCFS of the packet descriptor block.
6	IP%JWP	Make the PID created be permanent until the job logs out (if both bits IP%CPD and IP%JWP are on). Make the PID created be temporary until the process executes a RESET% monitor call (if bit IP%CPD is on and bit IP%JWP is not on). If bit IP%CPD is not on, bit IP%JWP is ignored.
7	IP%NOA	Do not allow other processes to use the PID created when bit IP%CPD is on. If bit IP%CPD is not on, bit IP%NOA is ignored.
8-17		Reserved for Digital.
18	IP%CFP	The packet is privileged. This bit can be set only by a process with WHEEL capability enabled. Refer to the _TOPS-20 Monitor Calls Reference Manual_ for a description of this bit.
19	IP%CFV	The packet is a page of 512 (decimal) words of data.
20	IP%CFZ	A zero-length message was sent.
21		Reserved for Digital.
22	IP%EPN	Page number in word .IPCFP of the packet descriptor block is 18 bits long
23		Reserved for Digital.
24-29	IP%CFE	Field for error code returned from <SYSTEM>INFO. Code Symbol Meaning 15 .IPCPI insufficient privileges 16 .IPCUF invalid function 66 .IPCKM PID has been deleted 67 .IPCSN <SYSTEM>INFO needs name 72 .IPCFF <SYSTEM>INFO free space exhausted 74 .IPCBP PID has no name or is invalid 75 .IPCDN duplicate name has been specified 76 .IPCNN unknown name has been specified 77 .IPCEN invalid name has been specified
30-32	IP%CFC	System and sender code. This code can be set only by a process with WHEEL capability enabled, but the monitor will return the code so a nonprivileged process can examine it. Code Symbol Meaning 1 .IPCCC Sent by <SYSTEM>IPCF 2 .IPCCF Sent by system-wide <SYSTEM>INFO

		3	.IPCCP	Sent by receiver's <SYSTEM>INFO
		4	.IPCCG	Sent by monitor for QUEUE% JSYS
33-35	IP%CFM	Field for special messages. This code can be set only by a process with WHEEL capability enabled, but the monitor will return the code so that a nonprivileged process can examine it.		
		Code	Symbol	Meaning
		1	.IPCFN	Process' input queue contains a packet that could not be delivered to intended PID.

7.3.2 PIDs

Any process that wants to send or receive a packet must obtain a PID. The process can obtain a PID by sending a packet to <SYSTEM>INFO requesting that a PID be assigned. The process must also include a symbolic name that is to be associated with the assigned PID.

The symbolic name can be a maximum of 29 characters and can contain any characters as long as it is terminated by a zero word. There should be mutual understanding among processes as to the symbolic names used in order to initiate communication. Once the name is defined, any process referring to that name must specify it exactly character for character.

Before a process can send a packet, it must know the receiver's symbolic name or PID. If only the receiver's name is known, the sender must ask <SYSTEM>INFO for the PID associated with the name, since all communication is via PIDs.

The association between a PID and a name is broken:

1. On a RESET% monitor call.
2. When the process is killed or the job logs off the system.
3. When a request to disassociate the PID from the name is made to <SYSTEM>INFO.

<SYSTEM>INFO will not allow a name already associated with a PID to be assigned again unless the owner of the name makes the request. Nor will <SYSTEM>INFO assign a PID once it has been used. This action protects against messages being sent to the wrong receiver by accident.

The PIDs of the sender and the receiver are indicated by words .IPCFS and .IPCFR, respectively, of the packet descriptor block.

7.3.3 Length and Address of Packet Data Block

Word .IPCFP of the packet descriptor block contains the length and the beginning address of the message. The length specified is one of two types, depending on the type of message (refer to Section 7.3.5). If the message is a short-form message, the length is the actual word length of the message. If the message is a long-form message, the length is 1000 (octal) words, that is, one page.

The address specified is either an address or a page number, depending on the type of message (refer to Section 7.3.5). When a message is sent, it is taken from this address. When a message is received, it is placed in this address.

7.3.4 Directories and Capabilities

Words .IPCFD and .IPCFC describe the sender at the time the message was sent and are used by the receiver to validate messages sent to it. These two words are not used when a message is sent, and if the sender of the packet supplies them, they are ignored. However, when a message is received, if the receiver of the packet has reserved space for these words in the packet descriptor block, the system supplies the appropriate values of the sender of the packet. The receiver of the packet does not have to reserve these words if it is not interested in knowing the sender's directories and capabilities.

7.3.5 Packet Data Block

The packet data block contains the message being sent or received. The message can be either a short-form message or a long-form message.

A short-form message is one to n words long, where n is defined by the installation. (Usually, n is assumed to be 10 words.) When a short-form message is sent or received, word .IPCFP of the packet descriptor block contains the actual word length of the message in the left half and the address of the first word of the message in the right half. A process always uses the short form when sending messages to <SYSTEM>INFO.

A long-form message is one page in length (1000 octal words). When a long-form message is sent or received, word .IPCFP of the packet descriptor block contains 1000 (octal) in the left half and the page number of the message in the right half. To send and receive a

long-form message, both the sender and receiver must have bit IP%CFV (bit 19) set in the first word of the packet descriptor block, or else an error code is returned.

7.4 SENDING AND RECEIVING MESSAGES

To send a message, the sending process must set up the first four words of the packet descriptor block. The process then executes the MSEND% monitor call. After execution of this call, the packet is sent to the intended receiver's input queue.

To receive a message, the receiving process must also set up the first four words of the packet descriptor block. The last two words for the directories and capabilities of the sender can be supplied, and the system will fill in the appropriate values. The process then executes the MRECV% monitor call. After execution of this call, a packet is retrieved from the receiver's input queue. The input queue is emptied on a first-message-in, first-message-out basis.

7.4.1 Sending a Packet

The MSEND% monitor call is used to send a message via IPCF. Messages are in the form of packets of information and can be sent to a specified PID or to the system process <SYSTEM>INFO. Refer to Section 7.5 for information on sending messages to <SYSTEM>INFO.

The MSEND% call accepts two words of arguments. The length of the packet descriptor block is given in AC1, and the beginning address of the packet descriptor block is given in AC2. Thus,

AC1: length of packet descriptor block. The length cannot be less than 4.

AC2: address of packet descriptor block

The packet descriptor block consists of the following four words:

.IPCFL	Flags
.IPCFS	Sender's PID
.IPCFR	Receiver's PID
.IPCFP	Pointer to packet data block containing the message being sent.

Refer to Section 7.3 for the details on the packet descriptor and packet data blocks.

The flags that are meaningful when sending a packet are described in Table 7-2. Refer to Table 7-1 for the complete list of flag bits.

Table 7-2: Flags Meaningful on a MSEND% Call

Bit	Symbol	Meaning
0	IP%CFB	Do not block process if no messages in queue; if set, error return if no messages.
1	IP%CFS	The sender's PID is given in word .IPCFS of the packet descriptor block.
2	IP%CFR	The receiver's PID is given in word .IPCFR of the packet descriptor block.
3	IP%CFO	Allow the sender to send one message above its send quota.
4	IP%TTL	Truncate message if larger than space reserved.
5	IP%CPD	Create a PID for the sender and return it in word .IPCFS of the packet descriptor block. The PID created is to be permanent and useable by other processes according to the setting of bits IP%JWP and IP%NOA.
6	IP%JWP	The PID created is to be job wide and permanent until the job logs out. If this bit is not on, the PID created is to be temporary until the process executes the RESET monitor call.
7	IP%NOA	The PID created is not to be used by other processes.
18	IP%CFP	The message being sent is privileged (refer to the TOPS-20 Monitor Calls Reference Manual).
19	IP%CFV	The message being sent is a long-form message (that is, a page). The page the message is being sent to cannot be a shared page; it must be a private page.
22	IP%EPN	Page number in word .IPCFP of the packet descriptor block is 18 bits long.

When bit IP%CFS is on in the flag word, the sender's PID is taken from word .IPCFS of the packet descriptor block. This word is zero if bit IP%CPD is on in the flag word, indicating that a PID is to be created for the sender. In this case, the PID created is returned in word .IPCFS.

When bit IP%CFR is on in the flag word, the receiver's PID is taken from word .IPCFR of the packet descriptor block. If this word is 0, then the receiver of the message is <SYSTEM>INFO. Refer to Section 7.5 for information on sending messages to <SYSTEM>INFO.

On successful execution of the MSEND% monitor call, the packet is sent to the receiver's input queue. Word .IPCFS of the packet descriptor block is updated with the sender's PID. Execution of the user's program continues at the second location after the MSEND% call. (MSEND%)

If execution of the MSEND% call is not successful, the message is not sent, and an error code is returned in AC1. The execution of the user's program continues at the instruction following the MSEND% call.

7.4.2 Receiving a Packet

The MRECV% monitor call is used to retrieve a message from the process' input queue. Before a process can retrieve a message, it must know if the message is a long-form message and also must set up a packet descriptor block.

The MRECV% monitor call accepts two words of arguments. The length of the packet descriptor block is given in AC1, and the beginning address of the packet descriptor block is given in AC2. Thus,

AC1: length of packet descriptor block. The length cannot be less than 4.

AC2: address of packet descriptor block

The packet descriptor block can consist of the following nine words. The last five words are optional, and if supplied by the receiver, the values of the sender will be filled in by the system.

.IPCFL	Flags
.IPCFS	Sender's PID
.IPCFR	Receiver's PID
.IPCFP	Pointer to packet data block where the message is to be placed.
.IPCFD	Connected and logged-in directories of the sender.
.IPCFC	Enabled capabilities of the sender.
.IPCSD	Number of sender's connected directory.
.IPCAS	Account string of sender.
.IPCLL	Byte pointer for destination of sender's node.

Refer to Section 7.3 for the details on the packet descriptor and packet data blocks.

The flags that are meaningful when receiving a packet are described in Table 7-3. Refer to Table 7-1 for the complete list of flag bits.

Table 7-3: Flags Meaningful on a MRECV% Call

Bit	Symbol	Meaning
0	IP%CFB	If there are no packets in the receiver's input queue, do not block the process and return an error code if the queue is empty. If this bit is not on, the process waits until a packet arrives, if the queue is empty.
1	IP%CFS	Use PID referenced in word .IPCFS as sender's PID.
2	IP%CFR	The receiver's PID is given in word .IPCFR of the packet descriptor block.
3	IP%CFO	Allow one send request above quota. (Default send quota is 2.)
4	IP%TTL	Truncate the message if it is larger than the space reserved for it in the packet data block. If this bit is not on and the message is too large, an error code is returned and no message is received.
5	IP%CPD	Create PID for sender and return in word .IPCFS.
6	IP%JWP	Make created PID job wide (ignored unless IP%CPD set).
7	IP%NOA	Do not allow other processes to use created PID (ignored unless IP%CPD set).
18	IP%CFP	Packet is privileged (requires IPCF capability enabled).

19	IP%CFV	The message is expected to be a long-form message (that is, a page). The page the message is being stored into cannot be a shared page; it must be a private page.
22	IP%EPN	Page number in word .IPCFP of the packet descriptor block is 18 bits long.

The information in word .IPCFS is not supplied by the receiver when the MRECV% call is executed. The system fills in the PID of the sender of the packet when the packet is retrieved.

Word .IPCFR is supplied by the receiver. If bit IP%CFR is on in the flag word, then the PID receiving the packet is taken from word .IPCFR of the packet descriptor block. If bit IP%CFR is not on in the flag word, then word .IPCFR contains either -1, to receive a packet for any PID belonging to this process, or -2, to receive a packet for any PID belonging to this job. When -1 or -2 is given, packets are not received in any particular order except that packets from a specific PID are received in the order in which they were sent. Any other values in this word cause an error code to be returned.

The information in words .IPCFD and .IPCFC is also not supplied by the receiver. If these two words have been specified by the receiver, the system fills in the information when the packet is retrieved. Word .IPCFD contains the sender's connected directory in the left half and the sender's logged-in directory in the right half. Word .IPCFC contains the enabled capabilities of the sender. These words describe the sender at the time the message was sent.

On successful execution of the MRECV% monitor call, the packet is retrieved and placed into the packet data block as indicated by word .IPCFP of the packet descriptor block. AC1 contains the length of the next packet in the queue in the left half and flags from the next packet in the right half (see below). This word returned in AC1 is called the associated variable of the next packet in the queue. If there is not another packet in the queue, AC1 contains zero. Execution of the user's program continues at the second instruction after the MRECV% call.

The flags returned in the right half of AC1 on successful execution of the MRECV% monitor call are described in Table 7-4.

Table 7-4: MRECV% Return Bits

Bit	Symbol	Meaning
30-32	IP%CFC	System and sender code, set only by a privileged process. The packet was sent by <SYSTEM>IPCF if the code is 1(.IPCCC). The packet was sent by the system-wide <SYSTEM>INFO if the code is 2(.IPCCF). The packet was sent by the receiver's <SYSTEM>INFO if the code is 3(.IPCCP).
33-35	IP%CFM	Field for return of special messages. If the field contains 1(.IPCFN), then the process' input queue contains a packet that was sent to another PID, but was returned to the sender because it could not be delivered.

If execution of the MRECV% call is not successful, a packet is not retrieved, and an error code is returned in AC1. The execution of the user's program continues at the instruction following the MRECV% call.

7.5 SENDING MESSAGES TO <SYSTEM>INFO

The <SYSTEM>INFO process is the central information utility for IPCF. It performs functions associated with names and PIDs, such as, assigning a PID or a name or returning a name associated with a PID.

A process can request functions to be performed by <SYSTEM>INFO by executing the MSEND% monitor call (refer to Section 7.4.1). The message portion of the packet (that is, the packet data block) sent to <SYSTEM>INFO contains the request being made. In other words, the total request to <SYSTEM>INFO is a packet consisting of a packet descriptor block and a packet data block containing the request.

Packet Descriptor Block

```
!=======================================================!
!                      flag word                        !
!-------------------------------------------------------!
!                    sender's PID                       !
!-------------------------------------------------------!
!                          0                            !
!-------------------------------------------------------!
!                 pointer to request                    !
!=======================================================!
```

Packet Data Block

```
!=======================================================!
!          code            !         function           !
!                          !                            !
!-------------------------------------------------------!
!                         PID                           !
!-------------------------------------------------------!
!                  function argument                    !
!=======================================================!
```

Refer to Section 7.4.1 for the descriptions of the words in the packet descriptor block. The receiver's PID (word .IPCFR) is 0 when sending a packet to <SYSTEM>INFO.

7.5.1 Format of <SYSTEM>INFO Requests

As mentioned previously, the packet data block (that is, the message portion) of the packet contains the request to <SYSTEM>INFO.

The first word (word .IPCI0) contains a user-defined code in the left half and the function being requested in the right half. The user-defined code is used to associate the response from <SYSTEM>INFO with the correct request. The functions that the process can request of <SYSTEM>INFO are described in Table 7-5.

The second word (word .IPCI1) contains a PID associated with a process that is to receive a duplicate of any response from <SYSTEM>INFO. If this word is zero, the response from <SYSTEM>INFO is sent only to the process making the request.

The third word (word .IPCI2) contains the argument for the function specified in the right half of word .IPCI0. The argument is different depending on the function being requested. The arguments for the functions are described in Table 7-5.

Table 7-5: <SYSTEM>INFO Functions and Arguments

Function	Argument	Meaning
.IPCIW	name	Return the PID associated with the given name (refer to Section 7.3.2 for the description of the name).
.IPCIG	PID	Return the name associated with the given PID.
.IPCII	name in ASCIZ	Assign the given name to the PID associated with the process making the request. The PID is permanent if IP%JWP was set in the flag word when the PID was originally created (refer to Table 7-1).
.IPCIJ	name in ASCIZ	Identical to .IPCII function.

7.5.2 Format of <SYSTEM>INFO Responses

Responses from <SYSTEM>INFO are in the form of a packet sent to the process that made the request. A copy of the response is sent to the PID given in word .IPCI1, if any.

The message portion (that is, the packet data block) of the packet contains the response from <SYSTEM>INFO. The format of this response is

```
!=======================================================!
!          code            !         function           !
!                          !                            !
!-------------------------------------------------------!
!                       response                        !
!-------------------------------------------------------!
!                       response                        !
!=======================================================!
```

The first word (word .IPCI0) contains the user-defined code in the left half and the function that was requested in the right half. These values are copied from the values given in the request.

The second and third words (words .IPCI1 and .IPCI2) contain the response from the function requested of <SYSTEM>INFO. The response is

different depending on the function requested. The responses from the functions are described in Table 7-6.

Table 7-6: <SYSTEM>INFO Responses

Function Requested	Response
.IPCIW	The PID associated with the name given in the request is returned in word .IPCI1.
.IPCIG	The name associated with the PID given in the request is returned in word .IPCI1.
.IPCII	No response is returned.

7.6 PERFORMING IPCF UTILITY FUNCTIONS

A process can request various functions to be performed by executing the MUTIL% monitor call. Some of these functions are enabling and disabling PIDs, creating and deleting PIDs, and returning quotas. Several of the functions that can be requested are privileged functions. These are described in the TOPS-20 Monitor Calls Reference Manual.

The MUTIL% monitor call accepts two words of argument. The length of the argument block is given in AC1, and the beginning address of the argument block is given in AC2.

The argument block has the following format:

```
!=======================================================!
!                    function code                      !
!-------------------------------------------------------!
!                argument for function                  !
!-------------------------------------------------------!
!                argument for function                  !
!=======================================================!
```

The arguments are different, depending on the function being requested. Any values resulting from the function requested are returned in the argument block, starting at the second word.

Table 7-7 describes the functions that can be requested, the arguments for the functions, and the values returned from the functions.

Table 7-7: MUTIL% Functions

Function	Meaning
.MUENB	Allow the PID given to receive packets. If the process executing the call is not the owner of the PID, the process must be privileged. Argument PID Value Returned None
.MUDIS	Disable the PID given from receiving packets. If the process executing the call is not the owner of the PID, the process must be privileged. Argument PID Value Returned None
.MUGTI	Return the PID associated with <SYSTEM>INFO. Argument PID or job number Value Returned PID of <SYSTEM>INFO
.MUCPI	Create a private copy of <SYSTEM>INFO for the specified job. The caller must have IPCF capability enabled. Argument PID to be assigned to <SYSTEM>INFO PID or number of job creating private copy
.MUDES	Delete the PID given. The process executing the call must own the PID being deleted. Argument PID to be deleted Value Returned None

.MUCRE — Create a PID for the process or job given. If the job number given is not that of the process executing the call, the process must be privileged. The flag bits that can be specified are IP%JWP and IP%NOA (refer to Table 7-1 for their descriptions).

Argument
flag bits in the left half, and process handle or job number in the right half

Value Returned
PID that was created

.MUSSQ — Set send and receive quotas for the specified PID. The caller must have IPCF capability enabled. The new send quota is given in bits 18-26, and the new receive quota is given in bits 27-35. The receive quota applies to the specified PID, but the send quota applies to the job to which that PID belongs.

Argumemts
PID
new quotas

.MUFOJ — Return the number of the job associated with the PID given.

Argument
PID

Value Returned
Job number associated with PID given

.MUFJP — Return all PIDs associated with the job given.

Argument
job number or PID belonging to the job

Values Returned
Two-word entries for each PID belonging to the job. The first word of the entry is the PID, and the second word has bits IP%JWP and IP%NOA set if appropriate (refer to Table 7-1 for the descriptions of these bits). The list of entries returned is terminated by a zero word.

.MUFSQ — Return the send quota and the receive quota for the PID given.

INTER-PROCESS COMMUNICATION FACILITY

Argument
PID

Values Returned
Send quota in bits 18-26 and receive quota in bits 27-35.

.MUFFP — Return all PIDs associated with the process of the PID given.

Argument
PID

Values Returned
Two-word entries for each PID belonging to the process. The first word of the entry is the PID, and the second word has bits IP%JWP and IP%NOA set if appropriate (refer to Table 7-1 for the descriptions of these bits). The list of entries returned is terminated by a zero word.

.MUSPQ — Set the maximum number of PIDs allowed for the specified job. The caller must have IPCF capability enabled.

Argument
job number or PID
PID quota

.MUFPQ — Return the maximum number of PIDs allowed for the job given.

Argument
Job number or PID belonging to the job

Value Returned
Number of PIDs allowed for the job given

.MUQRY — Return the packet descriptor block for the next packet in the queue of the PID given.

Argument
PID, -1 to return the next descriptor block for the process, or -2 to return the next descriptor block for the job

Values Returned
Packet descriptor block of next packet in queue.

.MUAPF — Associate the PID given with the process given.

Arguments
PID
process handle

Value Returned
None

.MUPIC — Place the specified PID on a software interrupt channel. An interrupt is then generated when:

1. The MUPIC function is issued while the PID has one or more messages in its receive queue.

2. The PID's receive queue changes its state from empty to containing a message. Subsequent entries to a queue that is not empty do not cause an interrupt.

If the channel number is given as -1, the PID is removed from its current channel.

The calling process and the process that owns the specified PID must belong to the same job.

Arguments
PID
channel number

.MUDFI — Set the PID of <SYSTEM>INFO. An error is given if <SYSTEM>INFO already has a PID. The caller must have IPCF capability enabled.

Arguments
PID of <SYSTEM>INFO

.MURSP — Return a PID from the system PID table. The PID is returned in word 2 of the argument block. The system PID table currently has the following entries:

0	.SPIPC	Reserved for Digital
1	.SPINF	PID of <SYSTEM>INFO
2	.SPQSR	PID of QUASAR
3	.SPMDA	PID of QSRMDA
4	.SPOPR	PID of ORION

Argument

index into system PID table

.MUMPS — Return the maximum packet size for the PID given.

Argument
PID

Value Returned
Maximum packet size for PID

.MUSKP — Set PID to receive deleted PID messages. Allows a controller task to be notified if one of its subordinate tasks crashes. After this function is performed, if the subordinate PID is ever deleted (via RESET or the .MUDES MUTIL function), the monitor will send an IPCF message to the controlling PID notifying it that the subordinate PID has been deleted. This message contains .IPCKP in word 0 and the deleted PID in word 1.

Argument
Source (subordinate) PID
Object (controller) PID

.MURKP — Return controlling PID for this subordinate PID.

Argument
Source (subordinate) PID
Object (controller) PID (returned)

On successful completion of the MUTIL% monitor call, the function requested is performed, and any value is returned are in the argument block. Execution of the user's program continues at the second location following the MUTIL% call.

If execution of the MUTIL% monitor call is not successful, no requested function is performed and an error code is returned in AC1. Execution of the user's program continues at the location following the MUTIL% call.

CHAPTER 8

USING EXTENDED ADDRESSING

The term "extended addressing" refers to the size of the addresses that TOPS-20 uses on the DECSYSTEM-20 Extended KL10 processor. Older versions of TOPS-20 (Release 4.1 and before) used 18-bit addresses; newer versions (Release 5 and after) use 30-bit addresses.

This chapter discusses the two main activities associated with using TOPS-20 monitor calls with extended addressing:

1. Writing new programs for execution in sections of memory other than section 0

2. Converting existing programs so that they can be executed in sections other than section 0

This chapter also contains information on hardware instructions and macros useful to MACRO programmers who use extended addressing.

The discussion in this chapter depends heavily on the material in the _DECsystem-10/DECSYSTEM-20 Processor Reference Manual_. Refer to that manual for a description of the format of 30-bit addresses, the algorithm the processor uses to calculate effective addresses, and the way that individual machine instructions work.

8.1 OVERVIEW

The TOPS-20 extended address space contains 32 (decimal) sections. Each section contains 512 pages of 512 words each (256K words). An 18-bit address, called a local address, can reference any word in a given section. A 30-bit, or global, address can reference any word in any section of memory.

In contrast, TOPS-20 V4.1 and earlier provided an 18-bit, 256K-word address space. The Program Counter (PC) register was 18 bits. For each instruction executed, the first action taken was the computation of an 18-bit effective address. The algorithm for calculating the

effective address (including indexing and indirecting rules) was the same for all instructions.

Because the TOPS-20 virtual address space is limited to 32 sections, and section numbers longer than 5 bits are illegal, legal addresses are effectively limited to 23 bits. When addressing data, you can view this 32-section address space as one large memory area.

From the point of view of program execution, however, memory is divided into 32 discrete sections. A program can have code in more than one section of memory, and it can execute that code (assuming the constraints discussed below), but it must change sections explicitly, as discussed below.

Compatibility for existing programs is provided by section 0. A program running in section 0 behaves as though it were being executed on a system without extended addressing, except for certain instructions such as XJRSTF. For more information on the actions of specific instructions, see the _DECsystem-10/DECSYSTEM-20 Processor Reference Manual_.

8.2 ADDRESSING MEMORY AND ACS

The extended format PC contains a section field and a word-within-section field. When an instruction is executed, only the word field is incremented. Column overflow is never carried from the word field to the section field. If the last word of a section is executed, and it is not a jump instruction, then the next instruction is fetched from word 0 of the same section. Thus a program can only change sections explicitly, by means of a PUSHJ, JRST, XJRST or XJRSTF instruction, and only an XJRST or an XJRSTF can change control from section 0 to another section.

Because a whole word cannot contain a 30-bit address and the program flags, the PC and flags are a two-word entity. The flag bits are in the first word, and the figure below represents the second word. Figure 8-1 shows the format of the address fields of the PC.

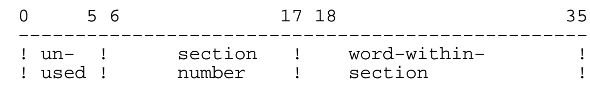


Figure 8-1: Program Counter Address Fields

The word (word-within-section) field consists of 18 bits and thus represents a 256K-word address space similar to the single-section address space of release 4 and earlier. The section number field is 12 bits, of which only the right-hand five bits can be nonzero because section numbers greater than 31 are illegal. The leftmost seven bits of the section number field must be zero. This provides room to address 32 separate sections, each of 256K words.

Each section is further divided into pages of 512 words, just as in earlier releases. The paging facilities allow the monitor to determine the existence and protection of each section.

The PC section field determines what section a program is running in. If the section field contains zero, the program is running in section 0. Most extended addressing features are not available to a program running in section 0. All quantities (including addresses), when calculated from section 0, are considered to be local (18 bits).

1. A program executing in section 0 cannot address memory in any other section. (One-word global byte pointers are an exception to this rule. Refer to Chapter 1 of the TOPS-20 Monitor Calls Reference Manual for more information.)

2. The program cannot jump from section 0 to another section unless it uses a monitor call or the XJRST or XJRSTF instruction.

3. The program runs exactly as it would run on a machine without extended addressing.

If the section field contains a number from 1 to 31 (decimal) inclusive, the program is executing in a nonzero section (a section other than section 0). The hardware considers addresses to be 30 bits, and the program can use extended addressing features.

A local address is defined as an 18-bit address in the same section as the program counter (PC) of the instruction. Local addresses are relative to the PC section. A global address is a 30-bit address, which therefore supplies its own section number.

The following paragraphs explain the way effective addresses are calculated in nonzero sections. In addition, see the description in the DECsystem-10/DECSYSTEM-20 Processor Reference Manual.

8.2.1 Instruction Format

The format of a machine instruction is the same as on an unextended machine. The effective address calculation depends on the address field (Y, 18 bits), the index field (X, 4 bits), and the indirect field (I, 1 bit). Figure 8-2 shows these fields.

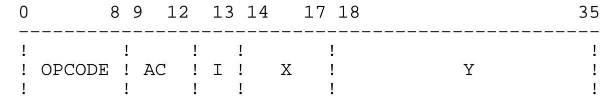

```
0       8 9  12  13 14   17 18                   35
---------------------------------------------------
!        !     !   !      !                       !
! OPCODE ! AC  ! I !   X  !           Y           !
!        !     !   !      !                       !
---------------------------------------------------
```

Figure 8-2: Instruction Word Address Fields

If I and X are 0, the instruction uses neither indexing nor indirection, so the effective address is Y (18 bits). The section number, since it is not specified in the address, is taken from the section field of the PC. The PC section field contains the number of the section from which the instruction was fetched. Such an 18-bit address is called a local address.

The following is an example of an instruction whose I, X and Y fields evaluate to an 18-bit effective address.

```
3,,400/    MOVEM T,1000
```

The effective address is word 1000 of the current section. The section from which the instruction is fetched is section 3, so the instruction moves the contents of register T into memory word 3,,1000.

8.2.2 Indexing

The first step in the effective address calculation is indexing. If the X field is nonzero, indexing is used. The calculation of the effective address then depends on the contents of the specified index register. Indexing may be local or global as follows:

- If the left half of the index register contains a negative number or zero, the contents of the right half (bits 18-35) are added to Y (from the instruction word) to yield an 18-bit local address.

  This is the way indexing is done on an unextended machine. It allows a program to use the usual AOBJN pointer and stack pointer formats for tables and stacks that are in the same section as the program. Note, however, that if the left half of the index register contains a positive number, the results are not the same.

- If the left half of the index register contains a positive number, the contents of bits 6-35 of the register are added to Y to yield a 30-bit global address.

This means that instructions can reference 30-bit (global) addresses by means of an index register. If the Y field is 0, the instruction refers to the address contained in X. The Y field can contain a positive or negative offset of magnitude less than 2^17.

8.2.3 Indirection

If the I field contains 1, the instruction specifies indirection. An indirect word is fetched from the address determined by Y and X. Two types of indirect word exist, Instruction Format Indirect Word (IFIW) and Extended Format Indirect Word (EFIW). They are described in the following section.

8.2.3.1 Instruction Format Indirect Word (IFIW) - This word contains Y, X, and I fields of the same size and in the same position as instructions (in bits 13-35). Bit 0 must be 1, and bit 1 must be 0; bits 2-12 are not used.

Figure 8-3 shows an instruction format indirect word.

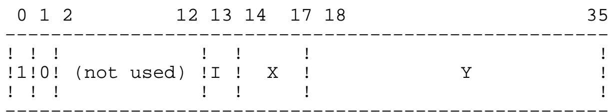


Figure 8-3: Instruction Format Indirect Word

The effective address calculation continues with the quantities in this word just as for the original instruction. Indexing can be specified and can be local or global depending on the left half of the index. Further indirection can also be specified.

Note that the default section for any local addresses produced from this indirect word is the section from which the word itself was fetched. This means that the default section can change during the course of an effective address calculation that uses indirection. The default section is always the section from which the last address word was fetched.

8.2.3.2 Extended Format Indirect Word (EFIW) - This word also contains Y, X, and I fields, but in a different format. Figure 8-4 shows an extended format indirect word.

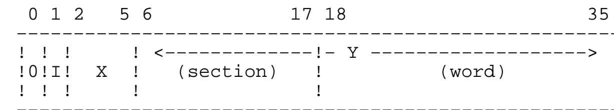


Figure 8-4: Extended Format Indirect Word

If indexing is specified in this indirect word (bits 2-5 nonzero), the contents of the entire index register are added to the 30-bit Y to produce a global address. This type of indirect word never produces a local address. The type of address calculation used does not depend on the contents of the index register specified in the X field.

Hence either Y or Y(X) can be used as an address or an offset within the extended address space, just as is done in the 18-bit address space. If further indirection is specified (bit 1 set), the next indirect word is fetched from Y as modified by indexing (if any). The next indirect word can be in instruction format or extended format, and its interpretation does not depend on the format of the previous indirect word.

8.2.3.3 Macros for Indirection - The system file MACSYM.MAC contains several convenient macros for constructing indirect words. Macro LFIWM generates local (instruction format) indirect words. Both the macros EP. and GFIWM may be used to generate global (extended format) indirect words.

8.2.4 AC References

A local address in the range 0-17 (octal) references the hardware ACs as memory. This is true in every section of memory.

A global address in section 1 in the range 1,,0 to 1,,17 (octal) also refers to the hardware ACs. A global address in any other section refers to memory. This means that the following behavior occurs.

1. Addresses in the range 0-17 reference ACs as expected. The instruction

```
MOVE 2,3
```

fetches the contents of hardware register 3 regardless of what section the instruction executes in.

2. To make a global reference to an AC, the global address must contain a section number of 0 or 1.

3. Arrays can cross section boundaries. Global addresses specifying any section except section 1 always refer to memory, never to the hardware ACs. For this reason, incrementing the address 6,,777777, for example, yields address 7,,000000, which is a memory location.

4. The ACs are regarded as local to any section; a local address (0-17) references the ACs from any section. Hence, a jump instruction which yields a local effective address of 0-17 in any section will cause code to be executed from the ACs.

8.2.5 Extended Addressing Examples

These instructions make local references within the current PC section:

```
3,,400/   MOVE T,1000              ; fetches from 3,,1000
          JRST 2000                ; jumps to 3,,2000
```

The following instructions scan table TABL, which is in the current section:

```
          MOVSI X,-SIZ
LP:       CAMN T,TABL(X)           ; TABL in current section
          JRST FOUND
          AOBJN X,LP
```

The following instructions scan table TABL, which is in section TSEC, by using a global address:

```
          MOVEI X,0
LP:       CAMN T,@[GFIWM TSEC,TABL(X)] ; extended format
          JRST FOUND
          CAIGE X,SIZ-1
          AOJA X,LP
```

Similarly, the EP. macro can be used for the same purpose:

```
          MOVEI X,0
LP:       CAMN T,@[EP.<TSEC>B17!TABL(X)]
          JRST FOUND
          CAIGE X,SIZ-1
          AOJA X,LP
```


USING EXTENDED ADDRESSING

The following examples illustrate various aspects of indexing and indirection in effective address calculation:

```
4/100
6,,1000/MOVE 1,@2000
6,,2000/LFIWM @4000
6,,4000/LFIWM 200(4)

Effective address = 300 in section 6


6,,SUB/   MOVE 1,@[LFIWM @ZZZ]

6,,ZZZ:   LFIWM @XXX
XXX:      LFIWM ARRAY(4)

Effective address = ARRAY+100 in section 6


6/14,,ADDRX
11,,ROU/MOVE 1,@[LFIWM (6)]

14,,ADDRX: LFIWM 100

Effective address = 14,,100
```

8.2.6 Immediate Instructions

Each effective address calculation yields a 30-bit address, defaulting the section if necessary. Immediate instructions use only the low-order 18 bits of this as their operand, however, and set the high-order 18 bits to 0. Hence instructions such as MOVEI and CAI produce identical results regardless of the section in which they are executed.

Two immediate instructions retain the section field of their effective addresses. These are:

- o XMOVEI (opcode 415) Extended Move Immediate
- o XHLLI (opcode 501) Extended Half Word Left to Left Immediate

8.2.6.1 XMOVEI - The XMOVEI instruction loads the 30-bit effective address into the AC, and sets bits 0-5 to 0. If no indexing or indirection is used, the number of the current section is copied from the PC to the AC. This instruction can replace MOVEI when a global address is needed.

The following example shows the use of the XMOVEI instruction in a subroutine call. The subroutine is in section XSEC, but the argument list is in the same section as the calling program.

```
XMOVEI AP,ARGLIST
PUSHJ P,@[GFIWM XSEC,SUBR]
```

The subroutine can reference the arguments with the following instruction.

```
MOVE T,@1(AP)
```

To construct the addresses of arguments, the subroutine can use the following instruction.

```
XMOVEI T,@2(AP)
```

The last two instructions assume that register AP contains the argument list pointer. If the address the calling program placed in AP is an IFIW, the section number in the effective address is that of the calling program. If the address the calling program placed in AP is an EFIW, the section number in the effective address of the argument block is determined by the section number the calling program placed in AP.

The argument list would be found in the caller's section because of the global address in AP. The section of the effective address is determined by the caller, and is implicitly the same as the caller if an IFIW is used as the arglist pointer, or is explicitly given if an EFIW is used.

8.2.6.2 XHLLI - The XHLLI instruction replaces the left half of the accumulator with the section number of the PC, and places zero in the right half of the AC. This instruction is useful for constructing global addresses.

8.2.7 Other Instructions

The instructions discussed here are affected by extended addressing, but not necessarily in the way that their effective addresses are calculated. In addition to the material presented here, see the DECsystem-10/DECSYSTEM-20 Processor Reference Manual regarding the following instructions: LUUOs, BLT, XBLT, XCT, XJRSTF, XJEN, XPCW, SFM.

8.2.7.1 Instructions that Affect the PC - These instructions are PUSHJ, POPJ, JRST. PUSHJ stores a 30-bit PC address, but stores no flags. It sets bits 0-5 of the destination word to 0.

POPJ restores a 30-bit PC address from the stack, but does not restore the flags. It also sets bits 0-5 of the destination word to 0.

The JSA and JRA instructions can be used only within a section. In section 0 the JSP and JSR instructions can store flags,,PC but then cannot transfer out of section 0. The JSP and JSR instructions can store flags,,PC in nonzero sections and then can transfer into any other section (if the address is specified with indexing or indirection).

8.2.7.2 Stack Instructions - PUSHJ, POPJ, PUSH, POP, and ADJSP. These instructions use a local or global address for the stack according to the contents of the stack pointer. Whether the stack address is local or global depends on the same rules as those that govern indexing in effective address calculation. (See section 8.2.2.) It is always best to use the ADJSP instruction when working with stacks. This instruction works in any section and will indicate when a pushdown overflow error occurs.

In brief, if the left half of the stack pointer is zero or negative (prior to incrementing or decrementing), the pointer references a local address and the address in its right half is the address of the current item in the stack. The stack pointer is incremented or decremented by adding or subtracting one from both sides, respectively.

If the left half of the stack pointer is positive, the entire word is taken as a global address. The stack pointer is incremented by adding 1, and decremented by subtracting 1.

A stack that contains global addresses can be used the same way a local stack is used. The global stack, however, can contain pointers to routines in other sections.

To protect against stack overflow and underflow, make the pages before and after the stack inaccessible. This method must be used because a global stack has no room for a count in the left half of the pointer.

8.2.7.3 Byte Instructions - To reference a byte in another section, you must use either a one-word, or a two-word, global byte pointer. Both global and local byte pointers are legal arguments to monitor calls from nonzero sections but there are some restrictions on the use of one-word global byte pointers from section 0. See Section 8.3 for further information.

Chapter 1 of the TOPS-20 Monitor Calls Reference Manual describes one-word global byte pointers. The DECsystem-10/DECSYSTEM-20 Processor Reference Manual describes two-word global byte pointers.

8.3 USING MONITOR CALLS

If a program runs in a single section, even though that section is not section 0, most monitor calls execute exactly the way they do in section 0. This is because when no section number is specified, the current section is the default.

The GTFDB% call, for example, requires that AC3 contain the address of the block in which to store the data it obtains from the file data block. This address can be an 18-bit address regardless of what section the monitor call is made from. When the monitor sees that the address is local, it obtains the section number from the PC of the process that makes the call.

The same is true of calls that accept page numbers. If a 9-bit page number is passed as an argument, the monitor obtains the section number from the PC of the process that made the call. Monitor calls arguments are discussed in Chapter 1 of the TOPS-20 Monitor Calls Reference Manual.

It is sometimes desirable to specify addresses in section 0 when executing a JSYS from a nonzero section. The bit PM%EPN for PMAP%, and FH%EPN for JSYSs that accept fork handles, prevent the current section (the section in which a program is running) from being the target section for the monitor call's arguments.

Another restriction on arguments passed to monitor calls executed in sections other than section 0 concerns universal device designators, which have the format 5xxxxx,,xxxxxx or 6xxxxx,,xxxxxx (.DVDES). Universal device designators are not legal except in section 0. This is because of the existence of one-word global byte pointers, which can have the same format.

Thus monitor calls that accept either a device designator or a byte pointer when called from section 0 do not accept universal device designators in any other section. Other device designators, such as .TTDES (0,,4xxxxx), can be used in any section. Conversely, these monitor calls that can accept either universal device designators or byte pointers do not accept one-word global byte pointers in section 0.

The calls SIR% and RIR% should not be used in sections other than section 0. These calls work in other sections only if all the code associated with these calls exists in the same section as the code that makes the call.

For example, if an SIR% call is executed in section 4, it executes correctly if and only if the code that generates the interrupts, the interrupt-processing routines, and all associated tables are also located in section 4. Thus, in programs intended to run in a section other than section 0, the XSIR% and XRIR% calls, described in Chapter 4, should be used in place of SIR% and RIR%. In general, it is recommended that the extended form of monitor calls be used since this form works in any section, including section 0.

8.3.1 Mapping Memory

The PMAP% monitor call accepts an 18-bit page number, half of which is a section number. Thus PMAP% can be used to map a page from one section to another. If the destination section does not exist, that section will be created.

The SMAP% monitor call maps one or more sections of memory. It works like the PMAP call, but maps sections instead of pages. If the destination section does not exist, SMAP% creates the section.

Access to the sections in a process map is determined by the same algorithm that determines access to a page within a given section. If a process section and a page in that section have different accesses, the access privileges are ANDed together. The process requesting access to the page gains access only if it has access rights at least equal to the ANDed protections.

For example, if a process has read access to a section and maps a page into that section for which the process has read and write access, the page is mapped, but the process gets only read access to the mapped page.

The following sections describe the SMAP% functions.

8.3.1.1 Mapping File Sections to a Process - This function maps one or more sections of a file to a process. All pages that exist in the source sections are mapped to the destination sections. Access to the mapped pages is determined by ANDing the access allowed to the file and the access specified in the SMAP% call.

Although files do not actually have section boundaries, this monitor call views them as having sections that consist of 512 contiguous pages. Each file section starts with a page number that is an integer multiple of 512.

This call cannot map a process memory section to a file. To map a process section to a file, use the PMAP% monitor call to map the section page-by-page.

This function of the SMAP% call requires three words of arguments, as follows:

AC1: source identifier: JFN,,file section number

AC2: destination identifier: fork handle,,process section number

AC3: flags,,count

The flags determine access to the destination section, and the count is the number of contiguous sections to be mapped. The count must be between 0 and 37 (octal). The flags are as follows:

B2(SM%RD)	Allow read access
B3(SM%WR)	Allow write access
B4(SM%EX)	Allow execute access
B18-35	The number of sections to map. This number must be between 1 and 37 (octal).

8.3.1.2 Mapping Process Sections to a Process - The SMAP% monitor call also maps sections from one process to another process. In addition, you can map one section of a process to another section of the same process. The SMAP% call maps all pages that exist in the source section to corresponding pages in the destination section.

If you map a source section into a destination section with SM%IND set, SMAP% creates the destination section using an indirect pointer. This means that the destination section will contain all pages that exist in the source section, and the contents of the destination pages will be identical to the contents of the source pages.

Furthermore, after SMAP% has mapped the destination section, changes that occur in the source section map cause the same changes to be made in the destination section map. This ensures that both the source section and the destination section contain the same data.

If SM%IND is not set, SMAP% creates the new section using a shared pointer. After SMAP% maps the destination section, changes that occur in the source section's map do not cause any change in the destination section's map. Thus after a short time the source and destination sections might contain different data.

If you request a shared pointer (SM%IND not set) to the destination section, what happens depends on the contents of the source section when the SMAP% call executes. The outcome is one of the following.

1. If the source section does not exist, the SMAP% call creates the section.

2. If the source is a private section, a mapping to the private section is established, and the destination process is co-owner of the private section.

3. If the source section contains a file section, the source section is mapped to the destination section.

4. If the source section map is made by means of an indirect section pointer, SMAP% follows that pointer until the source section is found to be nonexistent, a private section, or a section of a file.

This SMAP% function requires three words of arguments in AC1 through AC3.

AC1: Source identifier: fork handle,,section number

AC2: Destination identifier: fork handle,,section number

AC3: access flags,,the number of contiguous sections to map.

The number of sections mapped, the number in the right half of AC3, must be between 1 and 37.

The flags determine access to the destination section. The flags are as follows:

B2(SM%RD)	Allow read access
B3(SM%WR)	Allow write access
B4(SM%EX)	Allow execute access
B6(SM%IND)	Map the destination section using an indirect section pointer. Once the destination section map is created, the indirect section pointer causes the destination section map to change in exactly the same way that the source section map changes.
B18-35	Count of the number of contiguous sections to be mapped.

8.3.1.3 Creating Sections - Before you can use a nonzero section of memory, you must create it. If your program references a nonzero section of memory that does not exist (that is not mapped), the instruction that makes the reference fails.

This SMAP% function requires three words of arguments in AC1 through AC3, as follows:

AC1: 0

AC2: destination identifier: fork handle,,section number

AC3: flags,,count

The flags determine access to the destination section, and the count is the number of contiguous private sections to be created. This count must be between 1 and 37.

The flags in the left half of AC3 are as follows:

B2(SM%RD)	Allow read access
B3(SM%WR)	Allow write access
B4(SM%EX)	Allow execute access
B6(SM%IND)	Create the section using an indirect pointer
B18-35	The number of sections to create. This number must be between 1 and 37. All created sections are contiguous.

8.3.1.4 Unmapping a Process Section - You can use the SMAP% monitor call to unmap one or more sections of memory in a process. The contents of the section are lost.

If the section contains pages mapped from a file, this function does not cause the unmapped sections to be written back to the file from which they were mapped. Such pages must be mapped to the file by means of the PMAP% call.

This function requires three words of arguments in AC1 through AC3, as follows.

AC1: -1

AC2: Destination identifier: fork handle,,section number

AC3: 0,,count
The count is the number of contiguous sections to be unmapped. This number must be between 1 and 37.

USING EXTENDED ADDRESSING

8.3.2 Starting a Process in Any Section

You can use most of the calls described in Chapter 5 to control programs that run in a nonzero section. The SFORK% monitor call is an exception, and will not start a program in a nonzero section.

The XSFRK% monitor call starts a process in any section of memory. If the process is frozen (by means of the FFORK% call), XSFRK% changes the double-word PC, but does not resume execution of the process. To resume the execution of any frozen fork, use the RFORK% call.

The XSFRK% call requires three words of arguments in AC1 through AC3.

AC1: flags,,process handle

Flags:

SF%CON(1B0) continue a process that has halted. If SF%CON is set, the address in AC3 is ignored and the process continues from where it was halted.

AC2: PC flags,,0

AC3: address to which this call is to set the PC

The XSFRK% call also starts a process in section 0. To do so, set the left half of AC3 to zero and the right half of AC3 to the address in section 0 at which you want the process to start.

Most other calls consider an address with a zero in the left half to be a local address. The XSFRK% call, however, uses the contents of AC3 to set the PC. A PC with zero in the left half indicates an address in section 0.

8.3.3 Setting the Entry Vector in Any Section

The SEVEC% monitor call has room in its argument ACs for only a half-word address, so it cannot be used to set a process entry vector to an address in a nonzero section. The XSVEC% call, on the other hand, uses an AC for the address of the entry vector, and another AC for the length of the entry vector, and can specify an entry vector in any section of memory.

The XSVEC% call requires three words of arguments in AC1 through AC3.

AC1: process handle
AC2: length of the entry vector, or 0
AC3: address of the beginning of the entry vector

The length of the entry vector specified in AC2 must be less than 1000 words. If AC2 contains 0, TOPS-20 assumes a default length of two words.

8.3.4 Obtaining Information About a Process

Although the monitor calls described in Chapter 5 work in any section of memory, several of them can only return information about the section in which they are executed. The following paragraphs describe the monitor calls you can use to obtain information about any section of memory.

8.3.4.1 Memory Access Information - Several kinds of information about memory are important. Among them are whether a page or section exists (is mapped), and, if so, what the access to a page or section is. The RSMAP% and XRMAP% calls provide this information.

The RSMAP% monitor call reads a section map, and provides information about the mapping of one section of the address space of a process. RSMAP% requires one word of arguments in AC1: a fork handle in the left half, and a section number in the right half. It returns the access information in AC2.

The map information that RSMAP% returns in AC1 can be the following:

-1 no current mapping present (the section does not exist)

0 the mapping is a private section

n,,m where n is a fork handle or a JFN, and m is a section number. If n is a fork handle, the mapping is an indirect or shared mapping to another fork's section. If n is a JFN, the mapping is a shared mapping to a file section.

The access information bits returned in AC2 are the following:

B2(SM%RD) Read access is allowed

B3(SM%WR) Write access is allowed

B4(SM%EX) Execute access is allowed

B5(PA%PEX) The section exists

B6(SM%IND) The section was created using an indirect pointer.

USING EXTENDED ADDRESSING

Although the RSMAP% call does not return information on individual pages, the data it does return is useful in preventing error returns from the XRMAP% monitor call.

The XRMAP% call returns access information on a page or group of pages in any section of memory. Although the RMAP% call returns access data about a page in the current section, and you can use the RSMAP% call in any section of memory, you must use the XRMAP% call to obtain information about pages in any section other than the current section.

The XRMAP% call requires two words of arguments in AC1 and AC2.

AC1: process handle,,0

AC2: address of the argument block

The argument block addressed by AC2 has the following format:

```
!=====================================================!
!   Length of the argument block, including this word   !
!=====================================================!
! number of pages in this group on which to return data !
!-----------------------------------------------------!
!        number of the first page in this group         !
!-----------------------------------------------------!
!       address at which to return the data block       !
!=====================================================!
\                           .                           \
\                           .                           \
\                           .                           \
!=====================================================!
! number of pages in this group on which to return data !
!-----------------------------------------------------!
!        number of the first page in this group         !
!-----------------------------------------------------!
!       address at which to return the data block       !
!=====================================================!
```

The number of words in the argument block is three times the number of groups of pages for which you want access data, plus one. Each group of pages requires three arguments: the number of pages in the group, the number of the first page in the group, and the address at which the monitor is to return the access data.

Note that the address to which the monitor returns data should be in a section of memory that already exists. If it does not exist, the call will fail with an illegal memory reference.

The access information returned for each group of pages specified in the argument block is the following:

```
B2(RM%RD)   read access allowed
B3(RM%WR)   write access allowed
B4(RM%EX)   execute access allowed
B5(RM%PEX)  page exists
B9(RM%CPY)  copy-on-write access
```

For each page specified in the argument block that does not exist, XRMAP% returns a -1. It also returns a zero flag word for each such page. The data block to which XRMAP% returns the access information should therefore contain twice as many words as the number of groups of pages about which you want information.

If you execute an XRMAP% call to obtain information about a page in a nonexistent section, the XRMAP% call fails with an illegal memory reference. For this reason it is recommended to execute an RSMAP% call to determine that the section exists before you use XRMAP% to obtain information about any page within that section.

8.3.4.2 Entry Vector Information - To obtain the entry vector of a process in any section of memory, use the XGVEC% call. This call returns the length of the entry vector in AC2 and the address of the entry vector in AC3.

The XGVEC% call requires one word of argument: in AC1, the handle of the fork for which you want the entry vector.

8.3.4.3 Page-Failure Information - A page-fail word, described in the DECsystem-10/DECSYSTEM-20 Processor Reference Manual, contains information that allows a program to determine the cause of a page trap and the address of the instruction that caused the trap. This information allows a program to correct the cause of the page-fail trap. Once the program has corrected the cause of the page-fail trap, the program can continue execution.

The XGTPW% call obtains the page-fail word from the monitor's data base, and returns it to the calling program's address space. The XGTRP% call requires two words of arguments in AC1 and AC2.

AC1: process handle

AC2: address of the block in which to return data

8.3.5 Program Data Vectors

Program Data Vectors (PDVs) are data structures in a process that are known to the monitor by name and location. They contain information

about the program segments within a process. The PDV is a block of data that LINK writes into memory when loading and linking a program. The PDV resides in memory as a part of the program, and starts at a program data vector address (PDVA). PDVs are used to allow user programs to obtain information about other programs that can be mapped into a process. PDVs and PDVAs are manipulated by using the PDVOP% monitor call. (Refer to the TOPS-20 Monitor Calls Reference Manual for a complete description of the PDVOP% monitor call.) The PDVOP% monitor call can be used to obtain information about an execute-only process.

Certain words in the PDV (for example, .PVNAM) point to blocks of information. These words are in either IFIW or EFIW format (see Sections 8.2.3.1 and 8.2.3.2) except that they cannot use indexing, and any indirect chain pointed to by the word cannot go through an accumulator. This allows a program to find the address of a block pointed to by a PDV word by simply using an XMOVEI instruction. For example,

```
XMOVEI      AC1,@.PVNAM(AC2)
```

puts into AC1 the global address of the name of the PDV whose PDVA is in AC2.

8.3.5.1 Manipulating PDV Addresses - For the process specified in word .POPHD of the argument block, the .POGET function of the PDVOP% monitor call returns all PDVAs within the range of addresses specified in words .POADR and .POADE of the argument block. (See the description of the PDVOP% monitor call in the TOPS-20 Monitor Calls Reference Manual for the format of the argument block.) The address range supplied by words .POADR and .POADE determines which PDVAs are affected by any given call.

The .POADD function of the PDVOP% monitor call adds the PDVAs specified in the data block to the system's data base for the specified process. The PDVAs must be in ascending order within the data block.

The .POREM function of the PDVOP% monitor call removes a set of PDVAs from the system's data base for the specified process. Those removed are the ones within the range specified by words .POADR and .POADE of the argument block.

8.3.5.2 PDV Names - The .PONAM function of the PDVOP% monitor call returns the ASCIZ name of a PDV in memory. Word .POADR of the argument block must contain a valid PDVA for the specified process. The name returned is the one to which word .PVNAM of the PDV points. The string returned by .PONAM is placed into the data block.

For the specified process, the .POLOC function returns in the data block all the PDVAs of PDVs with the specified name. The byte pointer in AC3 points to the PDV name. Function .POLOC is affected by .POADR/.POADE words.

The following rules apply to the assignment of PDV names. If these rules are followed, it is quite unlikely that two packages that want to run in the same process will discover a conflict in PDV names.

1. PDV names assigned by DIGITAL will contain the character "%" at the end (or elsewhere). No PDV names assigned by users should contain the "%" character.
2. All PDV names containing the character "." are reserved to DIGITAL for future use.
3. The character "$" is reserved for a special use: PDV names of the form string1$string2 are reserved for the special class of use named by string1. Rules 1 and 2 still apply in this case.

As a general principle, avoid using PDV names that are insufficiently specific; use of such names invites conflicts with other packages.

8.3.5.3 Version Number - The .POVER function of the PDVOP% monitor call returns in the data block the version of a program in memory. Word .POADR must contain a valid PDVA for the specified process. The version returned is the one that word .PVVER of the PDV contains.

For more information on program data vectors, including explanations of the static memory map (pointed to by word .PVMEM) and the symbol table vector (pointed to by word .PVSYM), refer to the <u>TOPS-20 LINK Reference Manual</u>.

8.4 MODIFYING EXISTING PROGRAMS

Existing programs can be modified to run in any section of memory, including both section 0 and all other sections. The sections that follow discuss the changes that must be made to an existing program so that it runs in a single nonzero section.

8.4.1 Data Structures

Stacks, tables, and other data structures used in the past have often contained words with an address in the right half and a count in the left half. The count could be positive or negative because all

programs ran only in section 0, and when the contents of a word were evaluated for Effective Address calculation or address use in section 0, only the right half was considered.

In all other sections, the entire word is considered to be an address. If the left half of the word is negative, the left half is ignored when the address is evaluated, and the address is local. Thus for a word to contain an address in the right half and a count in the left half, the count must be negative.

8.4.1.1 Index Words - Be sure the left halves of index words contain a nonpositive quantity. To use the left half of an index register to hold a count, the count must be negative. If the left half is unused, it must be zero so that the effective address is a local address. If the left half contains a positive number, the indexed address will be global.

8.4.1.2 Indirect Words - To be sure that an indirect word in a nonzero section is evaluated as a local address, always set bit 0 of the indirect word. Argument lists that produce local addresses in section 0, for example, will produce local addresses in any section if bit 0 is set.

8.4.1.3 Stack Pointers - As mentioned above, the left halves of stack pointers must contain zero or a negative number to produce local addresses. A negative number in the left half is considered to be a count. A positive number in the left half is considered to be a section number.

8.5 WRITING MULTISECTION PROGRAMS

Multisection programs, programs that use more than one section of memory, are similar to single-section programs that run in nonzero sections. They allow you to place tables needed for processing interrupts in any section of memory (See Chapter 4), to use very large arrays, and to write modules of code that can be dynamically mapped into a section of memory and executed.

In a single-section program, local addresses and byte pointers are sufficient to specify any word or byte in the program's address space. In a multisection program, local addresses and byte pointers cannot specify any word or byte in the program's address space. Most monitor calls use only one AC per argument, so passing two-word global byte pointers is not possible. Thus, it is necessary to:

- o keep monitor call arguments in the same section of memory as the code making the call, or

- o use global arguments, or

- o use the global form of the monitor call.

In many multisection programs it is not necessary to keep all the arguments required by a call in the same section as the code that makes the call. Global arguments are required, and they take several forms. Chapter 1 of the *TOPS-20 Monitor Calls Reference Manual* gives details on the use of these arguments.

The following program computes a file checksum by XORing the words in all file pages. The program is loaded into section 0 and maps itself into section 1. It then jumps into section 1 to access the file data loaded into section 15.

```
        TITLE CHKSUM - COMPUTE A FILE CHECKSUM
        SEARCH MONSYM           ;STANDARD UNIVERSAL FILES
        SEARCH MACSYM
        .REQUIRE SYS:MACREL     ;GET JSERR SUPPORT ROUTINES

        STDAC.                  ;DEFINE STANDARD ACS

; PROGRAM CONSTANTS

        PDLSIZ==100             ;SIZE OF STACK
        CODSEC==1               ;SECTION TO MAP CODE INTO
        DATSEC==15              ;SECTION TO MAP FILE DATA INTO
        DATPAG==100             ;PAGE WITHIN DATSEC FOR FILE DATA
        PAGSIZ==1000            ;SIZE OF A PAGE

CHKSUM: RESET%                  ;RESET THE WORLD
        MOVE P,[IOWD PDLSIZ,PDL]
        MOVE T1,[.FHSLF,,0]     ;MAP THIS FORKS SECTION 0
        MOVE T2,[.FHSLF,,CODSEC] ;TO EXTENDED CODE SECTION
        MOVX T3,SM%RD!SM%WR!SM%EX!SM%IND+1
                                ;INDIRECT POINTER RD,WR,EX 1 SECTION
        SMAP%
         EJSHLT                 ;UNEXPECTED FATAL ERROR
GETFIL: SETZM FILJFN            ;SAY NO FILE SEEN
        TMSG <
ENTER FILE SPEC TO CHECKSUM: >  ;PROMPT USER FOR FILE
        MOVX T1,GJ%SHT!GJ%OLD!GJ%FNS ;OLD FILE
        MOVE T2,[.PRIIN,,.PRIOU] ;READ FILE SPEC FROM TERMINAL
        GTJFN%                  ;GET FILE SPEC
         ERJMPR BADFIL          ;CANNOT GET FILE TELL USER
        MOVEM T1,FILJFN         ;SAVE FILE JFN
        MOVX T2,FLD(^D36,OF%BSZ)!OF%RD
                                ;REQUEST READ ACCESS AND 36 BIT BYTES
        OPENF%                  ;OPEN THE FILE
```



```
         ERJMPR BADFIL          ;CANNOT OPEN FILE TELL USER

        XJRST [CODSEC,,DOCHKS]  ;ENTER EXTENDED SECTION
                                ;AND DO CHECKSUM

BADFIL: JSERR                   ;PRINT ERROR MESSAGE
        SKIPE T1,FILJFN         ;IS THERE A JFN
        RLJFN%                  ;YES. RELEASE IT
         EJSERR                 ;PRINT ERROR IF ANY
        JRST GETFIL             ;AND TRY TO GET FILE AGAIN

; THE FOLLOWING CODE RUNS IN A NONZERO SECTION AND
; DOES A CHECKSUM OF THE FILE STORED IN FILJFN

DOCHKS: SETZB Q1,Q2             ;Q1 HOLDS THE CHECKSUM.
                                ;INITIALLY ZERO
                                ;Q2 IS THE CURRENT FILE PAGE NUMBER
CHKLOP: MOVE T1,Q2              ;GET FILE PAGE NUMBER
        HRL T1,FILJFN           ;AND FILE JFN
        FFUFP%                  ;FIND FIRST USED PAGE
         ERJMPR NOPAGE          ;CAN'T GO ANALYZE ERROR
        HRRZ Q2,T1              ;REMEMBER CURRENT PAGE NUMBER
        AOS Q2                  ;USE NEXT HIGHER PAGE NEXT TIME
        MOVE T2,[<DATSEC>B26+DATPAG] ;THROUGH LOOP TO DATA PAGE
        HRLI T2,.FHSLF          ;IN DATA SECTION OF THIS FORK
        MOVX T3,PM%RD           ;READ ACCESS
        PMAP%                   ;MAP THE FILE PAGE
         EJSHLT                 ;UNEXPECTED FATAL ERROR
        HRLI T1,DATSEC          ;T1 IS INDEX INTO DATA PAGE.
        HRRI T1,DATPAG*PAGSIZ   ;SETUP SECTION AND PAGE ADDRESS
        MOVEI T2,PAGSIZ         ;T2 COUNTS THE WORDS IN A PAGE

; THE FOLLOWING LOOP DOES THE CHECKSUM FOR A PAGE

XORLOP: XOR Q1,(T1)             ;CHECKSUM THIS WORD
        AOS T1                  ;STEP TO NEXT WORD
        SOJG T2,XORLOP          ;DO THE WHOLE PAGE

        SETO T1,                ;UNMAP THE FILE PAGE
        MOVE T2,[<DATSEC>B26+DATPAG] ;TO DATA PAGE IN DATA
        HRLI T2,.FHSLF          ;SECTION OF THIS FORK
        MOVX T3,PM%RD
        PMAP%
         EJSHLT                 ;UNEXPECTED FATAL ERROR
        JRST CHKLOP             ;LOOP FOR MORE PAGES

; HERE WHEN FFUFP% FAILS

NOPAGE: CAIE T1,FFUFX3          ;NO USED PAGE FOUND?
        JSHLT                   ;NO. UNEXPECTED FATAL ERROR

; PRINT THE CHECKSUM AND QUIT
```

```
        TMSG <
THE FILE CHECKSUM IS: >
        MOVX T1,.PRIOU          ;PRINT IT ON THE TERMINAL
        MOVE T2,Q1              ;GET THE CHECKSUM
        MOVX T3,NO%MAG!FLD(^D8,NO%RDX)  ;UNSIGNED OCTAL NUMBER
        NOUT%
         EJSHLT                 ;UNEXPECTED FATAL ERROR

        MOVE T1,FILJFN          ;GET FILE AGAIN
        CLOSF%                  ;CLOSE IT
         EJSHLT                 ;UNEXPECTED FATAL ERROR

        HALTF%                  ;STOP PROGRAM
        XJRST [CHKSUM]          ;JUMP BACK TO SECTION 0 AND
                                ;START OVER IF USER CONTINUES

; STORAGE

PDL:    BLOCK PDLSIZ            ;STACK
FILJFN: BLOCK 1                 ;FILE JFN

        END CHKSUM
```


INDEX

-A-

AC, 1-2
  global reference, 8-7
  references, 8-6
Access
  copy-on-write, 5-5
  file, 3-2, 3-16
  file append, 3-16
  file frozen, 3-16
  file read, 3-16
  file restricted, 3-16
  file thawed, 3-16
  file unrestricted, 3-16
  file write, 3-16
  page, 5-5
Access bits
  OPENF%, 3-17
  PMAP%, 3-26
Accumulator (AC), 1-2
Accumulators, 1-3
  global reference, 8-7
  hardware, 8-6
  references, 8-6
Address
  global, 8-1, 8-6
  local, 8-4, 8-6
Address space, 8-1, 8-2
  process, 1-6, 5-1, 5-11
Addressing
  extended, 8-1
Addressing ACs, 8-2
Addressing memory, 8-2
ADJSP instruction, 2-2, 8-10
AIC% JSYS, 4-9, 4-17, 5-4
AOBJN pointer, 8-4
Argument block
  DEQ%, 6-14
  ENQ%, 6-8
  ENQC%, 6-15
  GTJFN% long form, 3-13
Arguments
  CFORK%, 5-8
  DEQ%, 6-12
  DIC%, 4-16
  ENQ% JSYS, 6-6
  ENQC%, 6-14
  GET%, 5-11
Arguments (Cont.)
  GTJFN% short form, 3-4
  IIC%, 5-19
  JFNS%, 3-33
  JSYS, 1-2, 1-3
  monitor calls, 1-3
  MRECV%, 7-9
  MSEND%, 7-7
  MUTIL%, 7-15
  OPENF%, 3-16
  PMAP%, 3-26, 3-28, 5-14
  PMAP% JSYS, 8-15
  RDTTY%, 2-9
  SFORK%, 5-15
  SIN%, 3-22
  SIR%, 4-6
  SMAP%, 3-29, 8-13, 8-14, 8-15
  SOUT%, 3-23
  XGTPW%, 8-19
  XRIR%, 4-15
  XRMAP% JSYS, 8-18
  XSFRK%, 8-16
  XSIR%, 4-7
  XSVEC% JSYS, 8-16
ASCII strings, 2-1, 3-21
ASCIZ pseudo-op, 1-6
ASCIZ strings, 2-1, 3-21
ATI% JSYS, 4-13

-B-

BIN% JSYS, 1-5, 3-21
  example, 1-5
Block
  packet data, 7-2
  packet descriptor, 7-2
BLT instruction, 8-9
BOUT% JSYS, 3-21
Byte, 2-1, 3-1
  reading a, 2-8
  transferring, 3-21
  writing a, 2-8
Byte instructions, 8-10
Byte manipulation instructions, 2-2
  ADJSP, 2-2
  IBP, 2-2
  ILDB, 2-2

Byte pointer, 8-10
  global, 8-10
  local, 8-10
  one-word global, 2-2, 8-10
  system standard for JSYS, 2-2
  two-word global, 2-2, 8-10

-C-

Calling sequence
  monitor calls, 1-3
Capability words, 5-11
CFORK% JSYS, 5-4, 5-8, 5-15, 5-19
  arguments, 5-8
  execution, 5-10
Changing sections, 8-2
Channel
  deactivating, 4-16
  panic, 4-5, 4-10, 4-11
Channel assignments
  software interrupt, 4-4
Channel table (CHNTAB), 4-7
CHNTAB, 4-7
CIS% JSYS, 4-17
Clearing interrupt system, 4-17
CLOSF% JSYS, 3-30
  example, 3-31
  execution, 3-31
  flag bits, 3-30
Closing a file, 3-30
Communication
  process, 1-6
Communication facility
  inter-process, 7-1
Control bits
  RDTTY%, 2-10
Control process, 1-6
Copy-on-write access, 5-5
Counter
  program, 8-1
Creating sections, 8-14

-D-

Data block
  packet, 7-2
Data transfer, 2-1
Data transfers, 3-19
Deactivating a channel, 4-16
Deadly embrace, 6-4, 6-5, 6-19
Deassigning terminal codes, 4-17
DEBRK% JSYS, 4-11
Deferred mode
  terminal interrupt, 4-14
Deleting inferior process, 5-20
DEQ% JSYS, 6-2, 6-6, 6-12
  argument block, 6-14
  arguments, 6-12
  functions, 6-12
Descriptor block
  packet, 7-2
Designator
  destination, 3-20
  primary input, 2-2, 3-20
  primary output, 2-2, 3-20
  source, 3-20
  universal device, 8-11
Destination designator, 3-20
Device designator
  universal, 8-11
DIC% JSYS, 4-16
  arguments, 4-16
DIR% JSYS, 4-16
Direct process control, 5-4
Disabling interrupt system, 4-16
DTI% JSYS, 4-17

-E-

Editing functions, 2-9
Effective address, 8-1
Effective address calculation,
    8-3, 8-8
  example, 8-8
    indexing, 8-8
    indirection, 8-8
  extended, 8-3
  immediate instructions, 8-8
  indexing, 8-5
  indirection, 8-5
  nonzero sections, 8-3
EFIW, 8-6, 8-20
EIR% JSYS, 4-9, 4-11, 4-17, 5-4
EJSERR macro, 1-5
EJSHLT macro, 1-5
ENQ quota, 6-3
ENQ% JSYS, 5-4, 6-2, 6-6, 6-17
  argument block, 6-8
  arguments, 6-6
  functions, 6-6
ENQC% JSYS, 5-4, 6-6, 6-14
  argument block, 6-15
  arguments, 6-14
  flag bits, 6-15

ENQUEUE/DEQUEUE (ENQ/DEQ)
    facility, 5-4, 6-1
  use of, 6-6
Entry vector, 8-16
  information, 8-19
EP. macro, 8-6, 8-7
ERCAL symbol, 1-4, 5-15
ERCALR symbol, 1-4
ERCALS symbol, 1-4, 1-5
ERJMP symbol, 1-4, 5-15
ERJMPR symbol, 1-4, 2-13
ERJMPS symbol, 1-4
Error returns
  monitor calls, 1-4
ERSTR% JSYS, 1-5
Execute-only process, 8-19
Extended addressing, 8-1, 8-3
  examples, 8-7
  using monitor calls with, 8-11
Extended format indirect word
    (EFIW), 8-6
Extended instruction format, 8-3
Extended page number, 8-11

-F-

FH%EPN, 8-11
File
  closing a, 3-30
  examples, 3-40
  opening a, 3-16
  pointer, 3-20
  reading from
    summary, 3-40
  referencing, 3-3
  sharing, 3-2, 6-1
  writing to
    summary, 3-40
File access, 3-2, 3-16
  codes, 3-2
File append access, 3-16
File frozen access, 3-16
File identifier, 3-2
File page mapping, 3-26
File pointer, 3-20
File read access, 3-16
File restricted access, 3-16
File section
  mapping, 8-12
File section mapping, 3-28
File specification, 3-3
  standard, 3-3
File thawed access, 3-16
File unrestricted access, 3-16
File write access, 3-16
Files, 3-1
Flag bits
  CLOSF%, 3-30
  ENQC%, 6-15
  GTJFN% long form, 3-14
  GTJFN% short form, 3-5
  MRECV%, 7-10
  MSEND%, 7-8
  SMAP%, 3-29
Flags
  packet descriptor block, 7-3
Format
  extended instruction, 8-3
  IPCF packet, 7-2
  <SYSTEM>INFO requests, 7-13
  <SYSTEM>INFO responses, 7-14
Format options
  JFNS%, 3-34
  NOUT%, 2-6
Functions
  DEQ%, 6-12
  ENQ%, 6-6
  MUTIL%, 7-15
  PDVOP%, 8-20
  RDTTY%, 2-9

-G-

GET% JSYS, 5-11, 5-14
  arguments, 5-11
GETER% JSYS, 1-5
GFIWM macro, 8-6
GFRKS% JSYS, 5-7
Global address, 8-1, 8-4, 8-6
Global byte pointer, 8-10
Global stack, 8-10
GNJFN% JSYS, 3-9, 3-36
  bits returned, 3-37
GTJFN% JSYS, 3-3, 3-4
  arguments
    long form, 3-12
    short form, 3-4
  bits returned, 3-10
  execution, 3-9, 3-14
  flag bits
    long form, 3-14
    short form, 3-5
  long form, 3-4, 3-12
    argument block, 3-13

GTJFN% JSYS (Cont.)
  short form, 3-4
    examples, 3-11
  summary, 3-15
GTSTS% JSYS, 3-31
  bits returned, 3-31

-H-

HALTF% JSYS, 2-8, 5-17
  example, 2-7
Handle section, 8-17
HFORK% JSYS, 5-17

-I-

I/O monitor calls, 2-2
IBP instruction, 2-2
Identifier
  file, 3-2
IFIW, 8-5, 8-20
IIC% JSYS, 4-10, 5-4, 5-19
  arguments, 5-19
ILDB instruction, 2-2
Illegal instruction trap, 1-4
Immediate instructions, 8-8
Immediate mode
  terminal interrupt, 4-14
Indexing, 8-4, 8-20
  example, 8-8
Indirection, 8-5, 8-20
  example, 8-8
  extended format indirect word (EFIW), 8-6
  instruction format indirect word (IFIW), 8-5
Inferior process, 1-6, 5-1
  characteristics, 5-8
  communicating with superior, 5-10
  creating, 5-8, 5-10
  deleting, 5-20
  parallel, 5-10
  starting, 5-15
  status, 5-17
  termination, 5-16
Information
  about process, 8-17
  entry vector, 8-19
  page-failure, 8-19
Initialization
  process, 2-8
Input
  terminal, 2-1
Input designator
  primary, 2-2
Instruction format
  extended, 8-3
Instruction format indirect word (IFIW), 8-5
Instructions
  byte, 8-10
  stack, 8-10
Inter-process communication facility
  receive quota, 7-1
  send quota, 7-1
  utility functions, 7-15
Inter-process communication facility (IPCF), 1-6, 5-4, 7-1
Interrupt, 4-1
  generating, 4-10
Interrupt channel assignments, 4-4
Interrupt channels
  activating, 4-9
Interrupt conditions, 4-4
Interrupt deferred mode
  terminal, 4-14
Interrupt dismissing, 4-11
Interrupt immediate mode
  terminal, 4-14
Interrupt processing, 4-10
Interrupt service routines, 4-6
Interrupt system
  clearing, 4-17
  disabling, 4-16
Interrupts
  terminal, 4-12
IPCF, 1-6, 5-4, 7-1
  packet data block, 7-2, 7-6, 7-12
    address, 7-6
    length, 7-6
  packet descriptor block, 7-2, 7-12
    flags, 7-3
  receive quota, 7-1
  send quota, 7-1
  utility functions, 7-15
IPCF packet format, 7-2

-J-

JFN, 3-1, 3-2
JFNS% JSYS, 3-33
  arguments, 3-33
  format options, 3-34
Job, 1-7
Job file number, 3-1, 3-2
Job structure, 1-6
  exapmle, 1-7
JRA instruction, 8-10
JRST instruction, 8-2, 8-9
JSA instruction, 8-10
JSP instruction, 8-10
JSR instruction, 8-10
JSYS, 1-2
  AIC%, 4-9, 4-17, 5-4
  arguments, 1-2, 1-3
  ATI%, 4-13
  BIN%, 1-5, 3-21
  BOUT%, 3-21
  CFORK%, 5-4, 5-8, 5-10, 5-15, 5-19
  CIS%, 4-17
  CLOSF%, 3-30
  DEBRK%, 4-11
  DEQ%, 6-2, 6-6, 6-12
  DIC%, 4-16
  DIR%, 4-16
  DTI%, 4-17
  EIR%, 4-9, 4-11, 4-17, 5-4
  ENQ%, 5-4, 6-2, 6-6, 6-17
  ENQC%, 5-4, 6-6, 6-14
  error returns, 1-4
  ERSTR%, 1-5
  GET%, 5-11, 5-14
  GETER%, 1-5
  GFRKS%, 5-7
  GNJFN%, 3-9, 3-36
  GTJFN%, 3-3, 3-4, 3-9
  GTSTS%, 3-31
  HALTF%, 2-8, 5-17
  HFORK%, 5-17
  I/O, 2-2
  IIC%, 4-10, 5-4, 5-19
  JFNS%, 3-33
  KFORK%, 5-4, 5-20
  MRECV%, 5-4, 7-7, 7-9
  MSEND%, 5-4, 7-7, 7-12
  MUTIL%, 5-4, 7-15
  NIN%, 2-4, 2-13, 2-14
  NOUT%, 2-5, 2-14
JSYS (Cont.)
  OPENF%, 3-2, 3-16
  PBIN%, 2-8, 2-14
  PBOUT%, 2-8, 2-14
  PDVOP%, 5-11, 8-19
  PMAP%, 3-25, 3-26, 3-27, 3-28, 5-11, 5-14, 5-15, 5-19, 8-12, 8-15
  PSOUT%, 2-3, 2-14
  RDTTY%, 2-5, 2-9, 2-12, 2-14
  RESET%, 2-8, 5-21, 7-5
  RFSTS%, 5-4, 5-17
  RFSTS% long form, 5-17, 5-18
  RFSTS% short form, 5-17
  RIN%, 3-24
  RIR%, 4-15, 8-11
  ROUT%, 3-24
  RSMAP%, 8-17
  SAVE%, 5-11
  SEVEC%, 8-16
  SFORK%, 5-4, 5-15
  SFRKV%, 5-16
  SIN%, 3-22
  SIR%, 4-6, 4-11, 5-4, 8-11
  SKPIR%, 4-14
  SMAP%, 3-28, 8-12
  SOUT%, 3-22, 3-23
  SPJFN%, 2-2
  SSAVE%, 5-11
  STIW%, 4-14
  WFORK%, 5-4, 5-16
  XGTPW%, 8-19
  XGVEC%, 8-19
  XRIR%, 4-15, 8-12
  XRMAP%, 8-18
  XSFRK%, 5-16, 8-16
  XSIR%, 4-6, 4-11, 4-17, 8-12
  XSVEC%, 8-16
JUMP instruction symbols, 1-4
  ERCAL, 1-4, 5-15
  ERCALR, 1-4
  ERCALS, 1-4, 1-5
  ERJMP, 1-4, 5-15
  ERJMPR, 1-4, 2-13
  ERJMPS, 1-4
JUMP instructions, 1-4

-K-

KFORK% JSYS, 5-4, 5-20

-L-

Level number
  resource, 6-4
LEVTAB, 4-8
LFIWM macro, 8-6
LINK, 8-19
Literals, 2-2
Local address, 8-4, 8-6
Local byte pointer, 8-10
Lock
  resource, 6-1
Long form GTJFN%, 3-12
LUUO instructions, 8-9

-M-

MACSYM, 1-3
MACSYM macros, 1-3
  EJSERR, 1-5
  EJSHLT, 1-5
  EP., 8-7
  indirection, 8-6
    EP., 8-6
    GFIWM, 8-6
    LFIWM, 8-6
  TMSG, 2-4
Mapping, 8-12
  file page, 3-26
  file section, 3-28
  file sections to a process, 8-12
  memory, 8-12
  page, 3-24, 5-14
  process page, 3-27
  process section, 8-13
  sections, 8-12
Memory, 8-2
Memory sharing, 5-5
Messages
  receiving process, 7-7
  sending process, 7-7
Monitor calls, 1-2
  arguments, 1-2, 1-3
  calling sequence, 1-3
  error returns, 1-4
  for processes, 5-7
  I/O, 2-2
  operation code, 1-2
MONSYM, 1-2, 2-3
MRECV% JSYS, 5-4, 7-7, 7-9
  arguments, 7-9

MRECV% JSYS (Cont.)
  execution, 7-11
  flagbits, 7-10
  flags returned, 7-11
MSEND% JSYS, 5-4, 7-7, 7-12
  arguments, 7-7
  execution, 7-9
  flag bits, 7-8
Multiple processes, 5-2
Multisection programs, 8-22
MUTIL% JSYS, 5-4, 7-15
  arguments, 7-15
  execution, 7-20
  functions, 7-15

-N-

NIN% JSYS, 2-4, 2-13, 2-14
  example, 2-7
NOUT% JSYS, 2-5, 2-14
  example, 2-6, 2-7
  format options, 2-6
Number
  reading a, 2-4
  writing a, 2-5

-O-

One-word global byte pointer, 2-2, 8-10, 8-11
OPENF% JSYS, 3-2, 3-16, 3-27
  access bits, 3-17
  arguments, 3-16
  examples, 3-19
Opening a file, 3-16
Operation code
  monitor calls, 1-2
Output
  terminal, 2-1
Output designator
  primary, 2-2
Ownership, 6-2, 6-17
  exclusive, 6-2, 6-17
  shared, 6-1, 6-2, 6-17

-P-

Packet, 7-1, 7-2
  receiving a, 7-9
  sending a, 7-7
Packet data block, 7-2, 7-6, 7-12
  address, 7-6

Packet data block (Cont.)
  length, 7-6
Packet descriptor block, 7-2, 7-12
  flags, 7-3
Packet format
  IPCF, 7-2
Page, 3-1
Page access, 5-5
Page mapping, 5-14
  file, 3-25
Page sharing, 5-5
Page-failure information, 8-19
Panic channel, 4-5, 4-10, 4-11
Parallel inferior processes, 5-10
PBIN% JSYS, 2-8, 2-14
PBOUT% JSYS, 2-8, 2-14
PC, 5-1, 8-1, 8-2, 8-9
  address, 8-9
  address fields, 8-2
PDV, 8-19
  names, 8-20
    rules, 8-20
PDVA, 8-19
  manipulating, 8-20
PDVOP% JSYS, 5-11, 8-19
  functions, 8-20
PID, 7-1, 7-5, 7-11
PM%EPN, 8-11
PMAP% JSYS, 3-25, 3-27, 3-28, 5-11, 5-14, 5-15, 5-19, 8-12, 8-15
  access bits, 3-26
  arguments, 3-26, 3-28, 5-14, 8-15
POINT pseudo-op, 2-1
Pointer
  file, 3-20
Pooled resources, 6-11
POP instruction, 8-10
POPJ instruction, 8-9, 8-10
.PRIIN symbol, 2-2, 2-9, 2-14, 3-20
Primary input designator (.PRIIN), 2-2, 3-20
Primary output designator (.PRIOU), 2-2, 3-20
Printing a string, 2-3
Priority level
  interrupt, 4-11
  software interrupt, 4-4

Priority level table (LEVTAB), 4-8
.PRIOU symbol, 2-2, 2-9, 2-14, 3-20
Process, 1-6, 1-7
  address space, 1-6, 5-11
  capabilities, 5-11
  communication, 1-6, 5-3, 5-19
  control, 1-6, 5-4
  deleting inferior, 5-20
  examples, 5-21
  execute-only, 8-19
  handle, 5-5, 5-10
  identifiers, 5-5
  inferior, 1-6, 5-1
  information about, 8-17
  JSYSs for, 5-7
  multiple, 5-2
  parallel, 5-1
  starting in any section, 8-16
  starting inferior, 5-15
  status word, 5-17
  structure, 1-6, 5-1
  superior, 1-6, 5-1
  terminating inferior, 5-16
  use of resources, 6-5
Process communication, 1-6, 5-3, 5-5, 5-19
  sharing pages, 5-19
  software interrupt, 5-4, 5-19
Process control, 5-4
Process handle, 5-5
Process ID (PID), 7-1, 7-5, 7-11
Process identifiers, 5-5
Process initialization, 2-8
Process mapping, 3-27
Process messages
  receiving, 7-7
  sending, 7-7
Process relationships, 5-1
Process section, 3-28
  unmapping, 8-15
Process status word, 5-17
Process structure, 1-6, 5-1
Process unmapping, 3-28
Program counter, 8-2
  address fields, 8-2
Program counter (PC), 5-1, 8-1, 8-9
  address, 8-9
Program data vector (PDV), 8-19
  address (PDVA), 8-19

Program data vector (PDV) (Cont.)
  manipulating PDVAs, 8-20
  names, 8-20
    rules, 8-20
  program version number, 8-21
Programs
  multisection, 8-22
Protection
  resource, 6-4
Pseudo-ops
  ASCIZ, 1-6
  POINT, 2-1
PSOUT% JSYS, 2-3, 2-14
  example, 2-7
PUSH instruction, 8-10
PUSHJ instruction, 8-2, 8-9, 8-10

-Q-

Queue, 6-1, 6-2
Quota, 7-1
  receive, 7-1
  send, 7-1

-R-

RDTTY% JSYS, 2-5, 2-9, 2-12, 2-14
  arguments, 2-9
  control bits, 2-10
  editing functions, 2-9
  example, 2-13
Reading a byte, 2-8
Reading a number, 2-4
Reading a string, 2-9
Reading from a file
  summary, 3-40
Receive quota, 7-1
Receiving a packet, 7-9
Referencing a file, 3-3
Releasing a resource, 6-12
RESET% JSYS, 2-8, 5-21, 7-5
  example, 2-7
Resource, 6-1
  level number, 6-4
  obtaining information about,
      6-14
  ownership, 6-2, 6-17
  pooled, 6-11
  protection, 6-4
  releasing a, 6-12
  requesting use of, 6-6
  sharing, 6-1, 6-17
Resource (Cont.)
  use by process, 6-5
Resource lock, 6-1
Resource name, 6-4
Resource ownership, 6-2
RFSTS% JSYS, 5-4, 5-17
  long form, 5-17, 5-18
    status-return block, 5-18
  process status word, 5-17
  short form, 5-17
RIN% JSYS, 3-24
RIR% JSYS, 4-15, 8-11
  example, 4-15
ROUT% JSYS, 3-24
RSMAP% JSYS, 8-17
  information returned, 8-17

-S-

SAVE% JSYS, 5-11
Section
  changing, 8-2
  creating, 8-14
  nonzero, 8-14, 8-16
  zero, 8-3, 8-11
Section handle, 8-17
Section mapping, 8-12
  file, 3-28
  file to process, 8-12
  process, 8-13
Sections, 8-2
Send quota, 7-1
Sending a packet, 7-7
SEVEC% JSYS, 8-16
SFM instruction, 8-9
SFORK% JSYS, 5-4, 5-15
  arguments, 5-15
SFRKV% JSYS, 5-16
Sharer groups, 6-17
  use of, 6-17, 6-18
Sharing files, 3-2, 6-1
Sharing pages, 5-19
Sharing resources, 6-1, 6-17
Short form GTJFN%, 3-4
  examples, 3-11
SIN% JSYS, 3-22
  arguments, 3-22
SIR% JSYS, 4-6, 4-11, 5-4, 8-11
  arguments, 4-6
SKPIR% JSYS, 4-14

SMAP% JSYS, 3-28, 8-12
  arguments, 3-29, 8-13, 8-14,
      8-15
  flag bits, 3-29
Software interrupt, 1-6, 4-10,
    5-19
  channel assignments, 4-4
  channels and priorities, 4-4
  disabling, 4-16
  dismissing, 4-11
  example, 4-18
  panic channel, 4-5, 4-10, 4-11
  priority level, 4-11
  priority levels, 4-4
  process communication, 5-4
  processing, 4-10
  service routines, 4-6
  tables, 4-6
Software interrupt system, 1-6,
    4-1, 5-16
  enabling, 4-9
  operational sequence, 4-2
  summary, 4-17
Source designator, 3-20
SOUT% JSYS, 3-22, 3-23
  arguments, 3-23
SPJFN% JSYS, 2-2
SSAVE% JSYS, 5-11
Stack
  address, 8-10
  global, 8-10
  pointer, 8-10
  register, 8-10
Stack instructions, 8-10
  ADJSP, 8-10
  POP, 8-10
  POPJ, 8-10
  PUSH, 8-10
  PUSHJ, 8-10
Standard file specification, 3-3
Starting a process, 8-16
Starting inferior process, 5-15
Status word
  process, 5-17
Status-return block, 5-18
STIW% JSYS, 4-14
String
  printing a, 2-3
  reading a, 2-9
Strings
  ASCII, 2-1, 3-21
  ASCIZ, 2-1, 3-21
Strings (Cont.)
  text, 2-1
  transferring, 3-22
    example, 3-23
Structure
  process, 1-6
Superior process, 1-6, 5-1
  communicating with inferior,
      5-10
<SYSTEM>INFO, 7-1, 7-5, 7-6, 7-7,
    7-9, 7-12
  functions and arguments, 7-13
  requests, 7-12
    format, 7-13
  responses, 7-14
<SYSTEM>INFO responses, 7-15

-T-

Table
  channel (CHNTAB), 4-7
  priority level (LEVTAB), 4-8
  software interrupt, 4-6
Terminal
  input, 2-1
  output, 2-1
Terminal codes
  deassigning, 4-17
Terminal interrupts, 4-12
  codes, 4-12
  deferred mode, 4-14
  generating, 4-13
  immediate mode, 4-14
Terminating inferior process,
    5-16
Text strings, 2-1
TMSG macro, 2-4
  example, 2-7
Transferring bytes, 3-21
Transferring data, 3-19
Transferring strings, 3-22
  example, 3-23
Trap
  illegal instruction, 1-4
Two-word global byte pointer, 2-2,
    8-10

-U-

Universal device designator, 8-11
Unmapping
  process page, 3-28

Unmapping (Cont.)
  process section, 8-15

-V-

Vector
  entry, 8-16
Virtual address space, 8-1
Virtual space, 1-6

-W-

WFORK% JSYS, 5-4, 5-16
Writing a byte, 2-8
Writing a number, 2-5
Writing to a file
  summary, 3-40

-X-

XBLT instruction, 8-9
XCT instruction, 8-9
XGTPW% JSYS, 8-19
  arguments, 8-19
XGVEC% JSYS, 8-19
XHLLI instruction, 8-8, 8-9
XJEN instruction, 8-9
XJRST instruction, 8-2
XJRSTF instruction, 8-2, 8-9
XMOVEI instruction, 8-8, 8-20
XPCW instruction, 8-9
XRIR% JSYS, 4-15, 8-12
  arguments, 4-15
XRMAP% JSYS, 8-18
  arguments, 8-18
XSFRK% JSYS, 5-16, 8-16
  arguments, 8-16
XSIR% JSYS, 4-6, 4-11, 4-17, 8-12
  arguments, 4-7
XSVEC% JSYS, 8-16
  arguments, 8-16