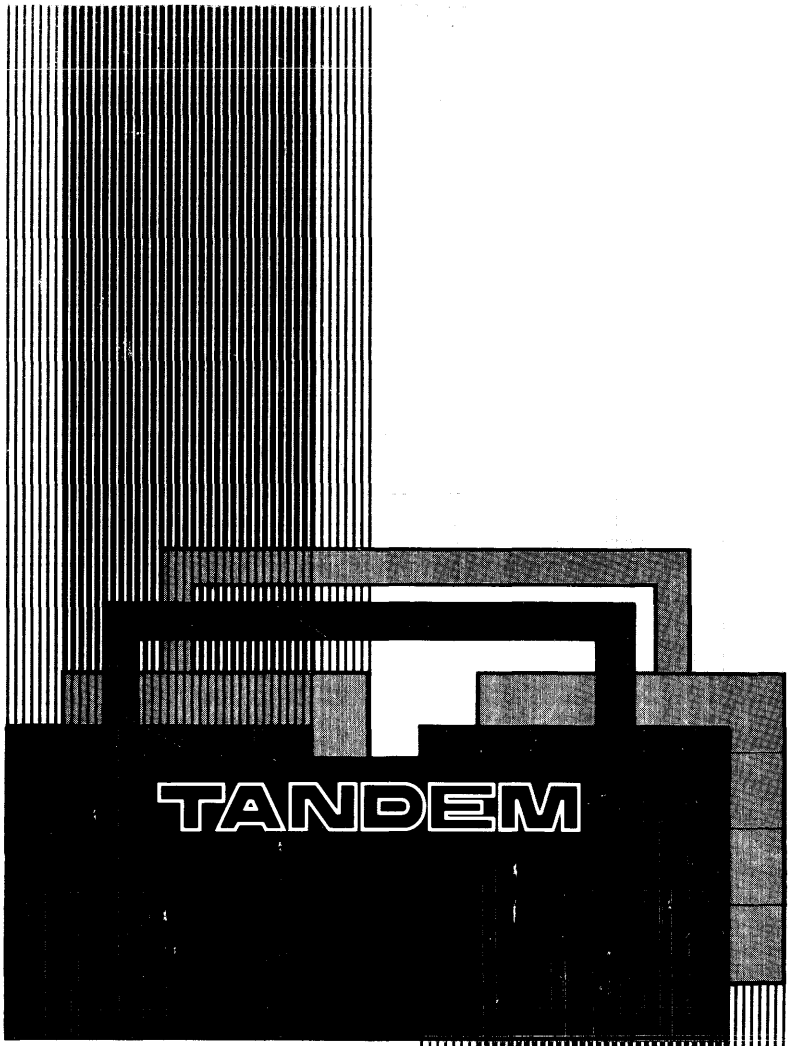


GUARDIAN Operating System Programming Manual

Volume 1

GUARDIAN (TM) OPERATING SYSTEM

PROGRAMMING MANUAL

Volume 1

Tandem Computers Incorporated
19333 Vallco Parkway
Cupertino, California 95014

P/N 82336 A00

April 1982
Printed in U.S.A.

PREFACE

This manual describes the interface between user programs and the GUARDIAN operating system on the Tandem NonStop and NonStop II systems.

Specifically, the manual discusses:

- calling the procedures provided by the GUARDIAN operating system for file management, process control, general utility, and checkpointing
- using traps and trap handling
- using the features provided for security of files and processes
- performing advanced memory management on NonStop systems and managing extended data segments on NonStop II systems
- using the sequential i/o procedures and the i/o formatter
- interfacing between application programs and the GUARDIAN command interpreter

This manual is for systems and applications programmers with special needs to call operating system procedures from their programs. Familiarity with the Tandem Transaction Application Language (TAL) or some other programming language, such as FORTRAN or COBOL, is recommended. Before using this manual, it is suggested that users read:

- Introduction to Tandem Computer Systems for a general overview of the system
- GUARDIAN Operating System Command Language and Utilities Manual, sections 1 and 2, for information about logging on to the system and running programs in general

The "advanced" subsections in sections 2, 5, and 8 discuss advanced features and require a knowledge of the system hardware registers, machine instructions, and/or operating modes.

For NonStop II systems only:

- NonStop II System Description Manual
- NonStop II System Operations Manual
- NonStop II System Management Manual
- GUARDIAN Operating System Messages Manual (NonStop II systems)
- DEBUG Reference Manual (NonStop II systems)

For both systems:

- GUARDIAN Operating System Command Language and Utilities Manual
- Transaction Application Language Reference Manual
- ENSCRIBE Programming Manual
- EXPAND Users Manual
- ENVOY Byte-Oriented Protocols Reference Manual
- ENVOYACP Bit-Oriented Protocols Reference Manual
- AXCESS Data Communications Programming Manual
- SORT/MERGE Users Guide
- Spooler/PERUSE Users Guide
- Spooler System Management Guide
- UPDATE/XREF Manual

For a combined index to subjects covered in Tandem technical manuals, identifying the manual and page number for each reference, refer to the following publications:

- Master Index (NonStop systems)
- Master Index (NonStop II systems)

For a complete list of technical manuals and manual part numbers for Tandem NonStop systems and Tandem NonStop II systems, refer to the following publication:

- Technical Communications Library

CONTENTS

Volume 1

FIGURES

Volume 1

SYNTAX CONVENTIONS IN THIS MANUAL

The following is a summary of the characters and symbols used in the syntax notation in this manual.

NOTATION	MEANING
UPPER-CASE CHARACTERS	All keywords and reserved words appear in capital letters. (A keyword is defined as one that, if it is present at all in the context being described, must be spelled and positioned in a prescribed way, or an error will result. A reserved word is one that can only be used as a keyword.) If a keyword is optional, it is enclosed in brackets. If a keyword is required, it is underlined.
<lower-case characters>	All variable entries supplied by the user are shown in lower-case characters and enclosed in angle brackets. If an entry is optional, it is enclosed in brackets. If an entry is required, it is underlined.
Brackets	Brackets, [ ], enclose all optional syntactic elements. A vertically-aligned group of items enclosed in brackets represents a list of selections from which one, or none, may be chosen.
Braces	A vertically-aligned group of items enclosed in braces, { }, represents a list of selections from which exactly one must be chosen.
Ellipses	An ellipsis (...) following a pair of brackets that contains a syntactic element preceded by a separator character indicates that that element may be repeated a number of times. An ellipsis following a pair of braces that contains a series of syntactic elements preceded by a separator character indicates that the entire series may be repeated, intact, a number of times. (NOTE: In coding syntax of this

<parameters> are described as follows:

<parameter>,<type> : {ref / value} [: <num elements>],

<type> is INT, INT(32), or STRING

"ref" indicates a reference parameter. Note that if a parameter is a "STRING:ref" parameter, a word-addressed variable (e.g., INT) can be passed for that parameter; the TAL compiler will produce instructions to convert the word address to a byte address. Note, however, that on NonStop systems, an invalid address will result if the word address is greater than 32767.

<num elements> indicates that the procedure returns a value of <type> to <parameter> for <num elements>. An asterisk "*" in this position indicates that the number of elements returned varies depending on the number of elements requested.

"value" indicates a value parameter.

SECTION 1

INTRODUCTION TO THE GUARDIAN OPERATING SYSTEM

The basic design philosophy of the Tandem NonStop and NonStop II Systems is that no single module failure will stop or contaminate the system. This capability is referred to by Tandem Computers as "NonStop" operation.

Overseeing NonStop system operation is the Tandem GUARDIAN operating system. The GUARDIAN software provides the multiprocessing (parallel processing in separate processor modules), multiprogramming (interleaved processing in one processor module), and NonStop capabilities of the system.

In a typical system, master copies of the GUARDIAN operating system, configured for the specific application, are kept in a "system" area (for NonStop systems) or a specially named "SYSnn" subvolume (for NonStop II systems) on a "mirrored" disc volume. (See figure 1-1.) Critical and frequently used parts of the GUARDIAN operating system are resident (i.e., always present) in each processor module's memory. As such, the system's capabilities are maintained even if a processor module, i/o channel, or disc drive fails. Non-critical or less frequently used parts of the GUARDIAN operating system are virtual, and are brought into a processor module's memory from disc only when needed.

- The capability for processes to communicate with each other regardless of the processor module where they are executing

- Providing the virtual memory function by automatically bringing absent memory pages in from disc when needed

- Scheduling processor module time among multiple processes according to their application-assigned priorities (a "process" is an executing program)

The GUARDIAN operating system provides an additional and extremely important function. Concurrent with application program execution, the GUARDIAN operating system continually checks the integrity of the system. This is accomplished as follows: The GUARDIAN operating system in each processor module at a predefined interval transmits "I'm alive" messages to the GUARDIAN operating system in every processor module (this interval is typically one second). Following this transmission, the GUARDIAN operating system in in each processor module checks for receipt of an "I'm alive" message from every other processor module. If the operating system in one processor module finds that a message has not been received from another processor module, it first verifies that it can transmit a message to its own processor module. If it can, it assumes that the non-transmitting processor module is inoperative; if it can't, it takes action to ensure that its own module does not impair the operation of other processor modules. In either case, the operating system then informs system processes and interested application processes of the failure.

An application program "sees" operating system services as a set of library procedures. The library procedures have names such as "READ", "WRITE", "OPEN", etc. To request an operating system service (e.g., input), a call to the appropriate operating system procedure is written in the application program (e.g., "READ"). (The operating system library procedures exist in the system code area and therefore are shared by all processes).

The operating system services that can be requested programmatically or that affect application program design are categorized as follows (overviews of each of these services are given in the remainder of this section):

- Process Control (run, suspend, and stop programs). Process control services are described in detail in

  - Section 3. PROCESS CONTROL

- File system (perform input/output operations). File system services are described in detail in

  - Section 2. FILE SYSTEM

- System Messages (communicate information from the GUARDIAN operating system to application processes). System messages are described in detail in

PROCESS CONTROL

A "process" is the execution of a program under control of the GUARDIAN operating system. It is the basic executable unit known to the operating system. Specifically, the term "program" indicates a static group of instruction codes and initialized data -- the output of a compiler; the term "process" denotes the dynamically changing states of an executing program. The same program file can be executing concurrently a number of times; each execution is a separate process.

The executing environment of a given process is a single processor module (the processor module where a process executes is specified at run time). A process's environment consists of a code area, containing instruction codes and program constants, and a separate data area, containing variables and hardware environment information. A given code area is shared by all processes that are executing the same program file. This is permissible because information within the code area cannot be modified. Each process, however, has its own separate, private data area.

The following terms referring to processes are used throughout this manual (for a more complete explanation, refer to section 3, "Process Control"):

- Process Creation

  The term "process creation" refers to the action performed by a special system process called the "System Monitor" when a program is initially prepared for execution. Process creation is initiated by application programs or by the Command Interpreter (COMINT) through the process control NEWPROCESS procedure.

  When the Command Interpreter is used to run a program, a "startup" interprocess message is sent to the newly created process. This message contains default disc volume and subvolume names, names of input and output files, and any application-dependent parameters specified through the RUN command. The startup message can be read by the new process via standard GUARDIAN file management procedures. (See "Interprocess Communication" in section 2.9.)

- Creator

  Another term, "creator", refers to the process that initiated a process creation (by calling the NEWPROCESS procedure). For example, the Command Interpreter is the "creator" of processes it starts when the RUN command is given.

  Certain attributes are associated with being a creator:

  - A creator receives a notification if a process it has created is deleted.

Process Structure

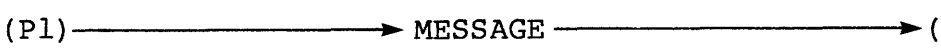
The process structure provided by the GUARDIAN operating system allows a program to be written as though it could run on a processor of its own. This abstraction is possible because

- each process executes independently of and without interference from all other processes
- each process's environment is private from all other processes

The process structure allows program functions (whether they are operating system or application functions) to be modularized. Modules can be written and tested independently of other modules. If a module is known to execute correctly when run by itself, it will be assured of running when run concurrently with other modules.

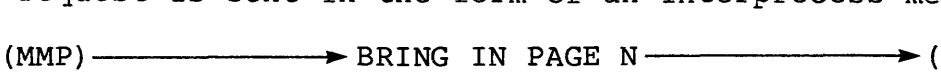
The GUARDIAN operating system is essentially a collection of processes, each process performing a specific function. For example, a memory manager process provides the virtual memory function for its processor; an i/o process (of which there are many) controls one or more similar i/o devices.

Processes communicate information among one another via messages.



(P) = process

For example, a GUARDIAN memory manager process may request that a GUARDIAN disc i/o process bring an absent memory page in from disc. The request is sent in the form of an interprocess message:



Applications are structured in much the same way as the operating system. That is, specific functions are performed by independent processes which communicate with each other via interprocess messages.

A common structure for applications is the "requestor/server" process relationship. With this structure, one or more "requestor" processes make requests of a common "server" process (an application may consist of several of the requestor/server relationships). A request is made in the form of an interprocess message (sent via the file system). The server makes a reply to the message via the file system (the reply usually consists of the requested data).

(RP) = requestor process (SP) = server process

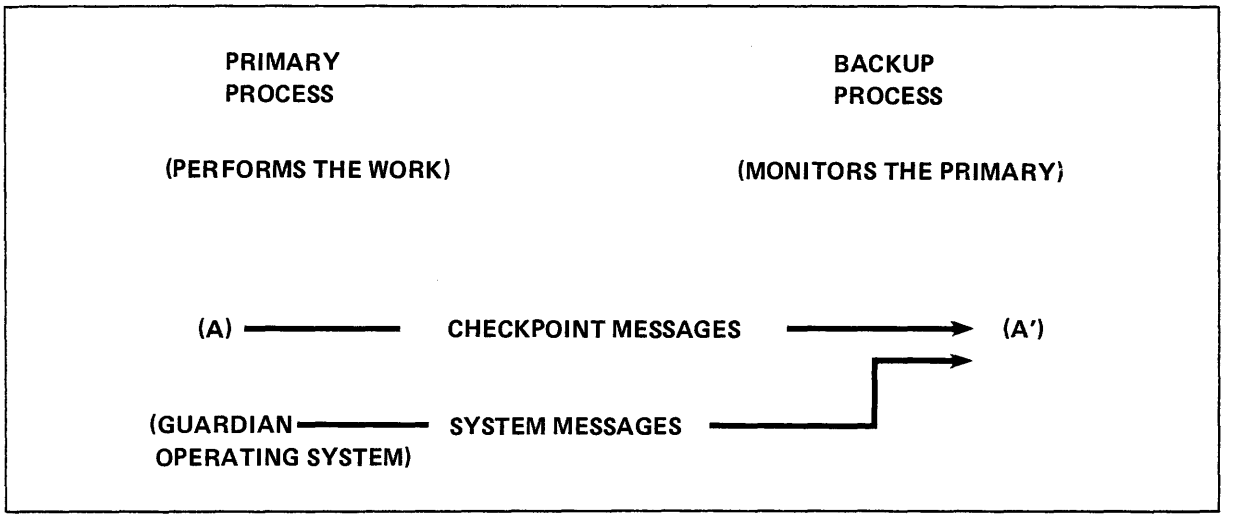


Figure 1-2. A Primary/Backup Process Pair

A process pair is typically identified by a single process name. A process pair's process name is entered into the Process-Pair Directory (PPD) when the first process of the pair is created. Also at this time, the identity of the "ancestor process" is entered into the PPD. (An "ancestor process" is the process responsible for creation of the first member of a process pair). The PPD provides capabilities that are useful for NonStop programming. For example, one member of a process pair is notified if the other member stops executing; the ancestor process is notified when the process name is deleted from the PPD (the latter occurs when the last process associated with a process name stops or fails). There are also NonStop aspects of communicating with named process pairs (see "File System", section 2).

Process Control Functions

Process control operations are performed by calling the GUARDIAN process control procedures. These procedures include:

NEWPROCESS	creates a process (runs a program) and, optionally, gives it a name (if a name is given, the name is entered into the Process-Pair Directory)
MYTERM	provides the file name of a process's home terminal
DELAY	suspends the calling process
PRIORITY	changes the calling process's execution priority
STOP	deletes a process with a normal indication
ABEND	deletes a process with an abnormal indication

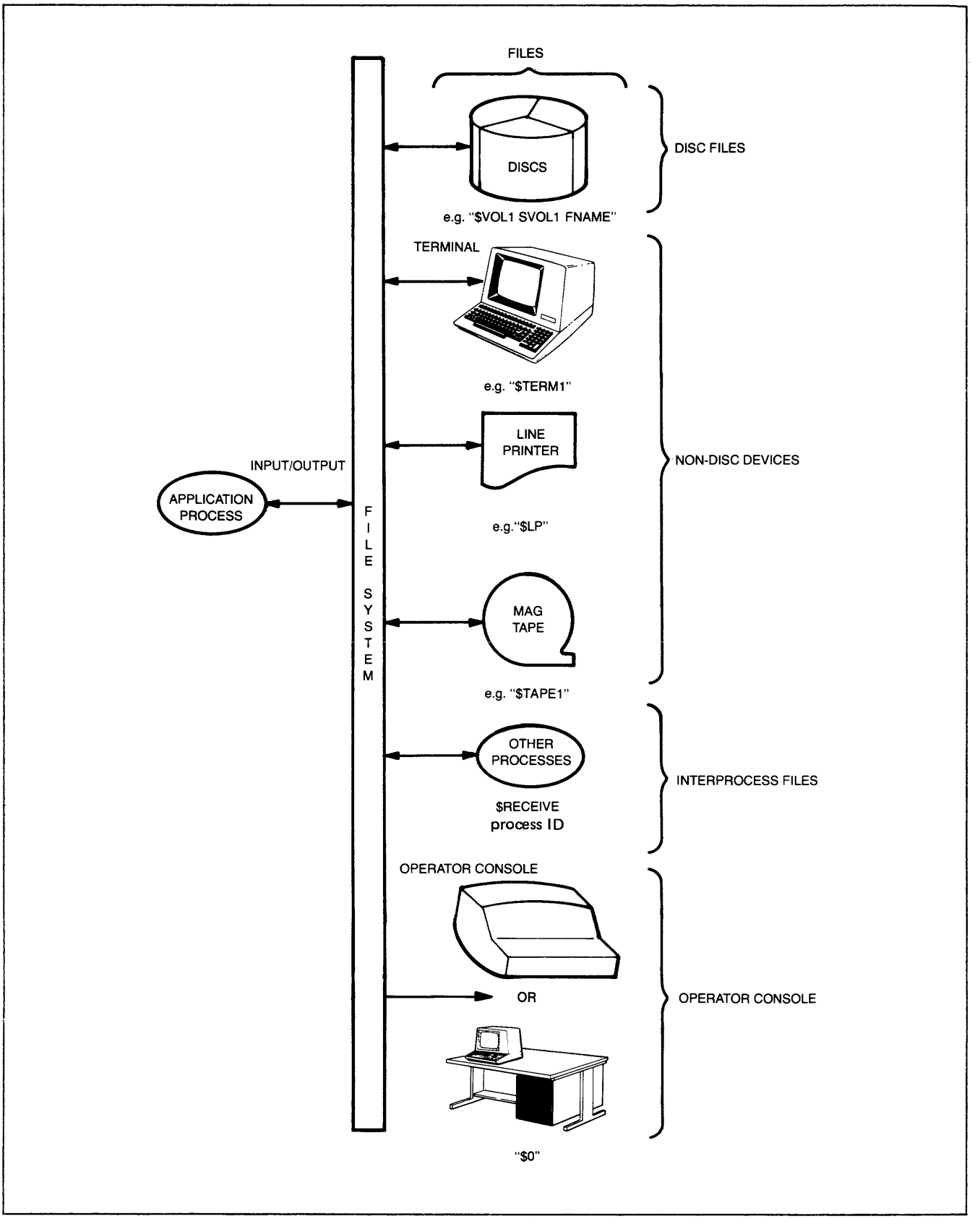


Figure 1-3. Files

Then to write (output) to the file, the file system WRITE procedure might be called in the following manner:

```
CALL WRITE(filenumber,buffer,count);
```

"buffer" is an array in the program's data area containing the information to be written. "count" is the number of bytes to be written.

Or to read (input) from the same file:

```
CALL READ(filenumber,buffer,count,numread);
```

Several other procedures are provided for performing device-dependent operations.

UTILITY PROCEDURES

As part of the operating system, procedures are provided to perform utility operations. These include:

DEBUG	calls the system debug facility
FIXSTRING	is used to edit a string of characters based on information supplied in an editing template
HEAPSORT	sorts an array of equal-size elements in place
INITIALIZER	reads the startup message and, optionally, the ASSIGN and PARAM messages to prepare global tables and initialize File Control Blocks (FCB's)
LASTADDR	provides the global ('G'[0] relative) address of last word in the application's data area
NUMIN	converts the ASCII representation of a number into its binary equivalent
NUMOUT	converts the internal machine representation of a number to its ASCII equivalent
TIME	provides the current date and time

SYSTEM MESSAGES

The operating system sends messages directly to application processes to inform the application of certain system conditions. These are referred to as "system messages". System messages are read using the GUARDIAN file system procedures. Examples of system messages are:

- CPU Down - processor module failed.

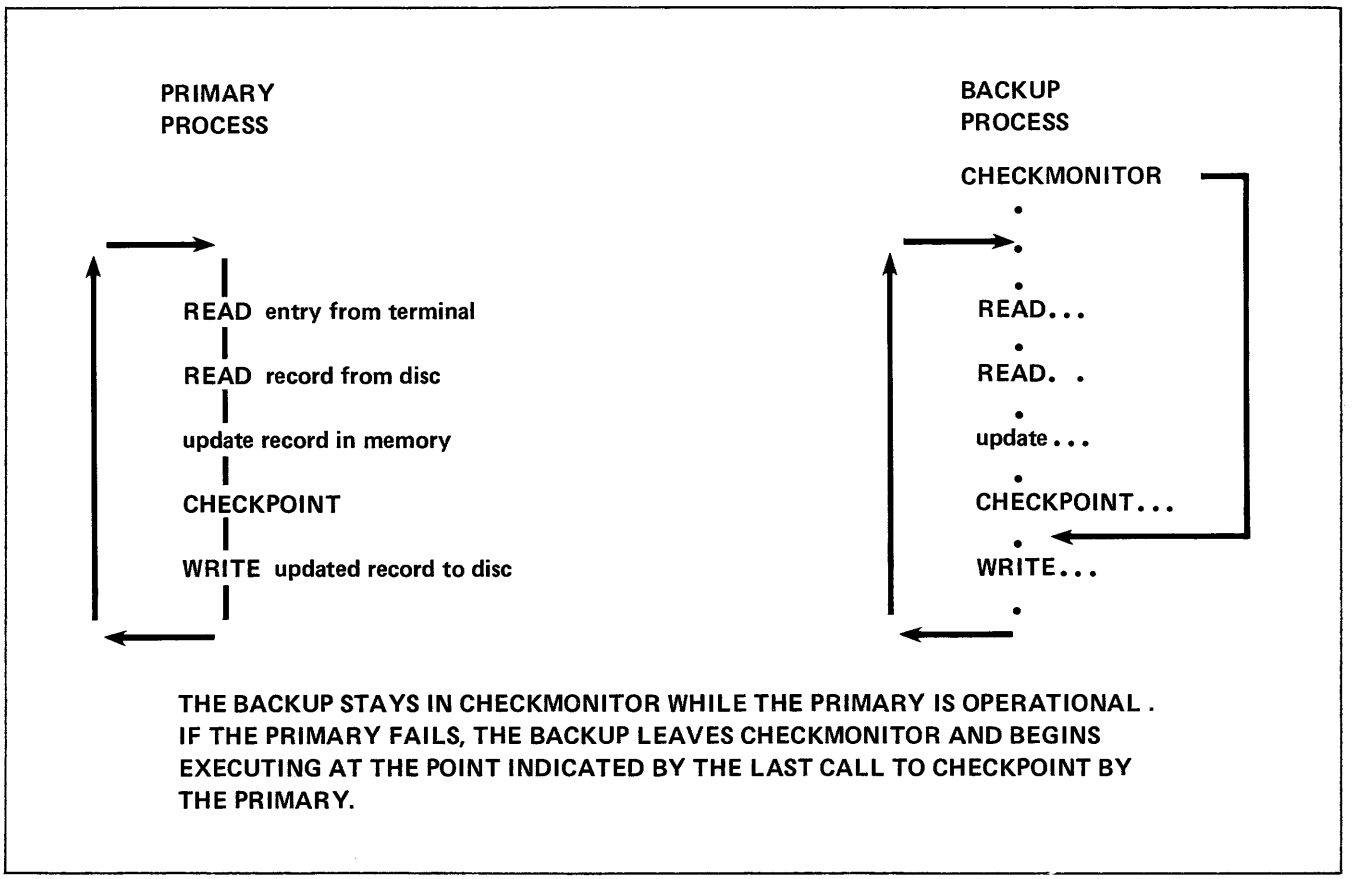


Figure 1-4. Checkpointing

When the checkpointing facility is used, each process in a process pair has the same set of files open, as shown in figure 1-5. This ensures that the backup process has immediate access to the files in the event of the primary's failure.

- Uncorrectable Memory Error
- Map Parity Error (NonStop systems only)

Generally, the first four trap conditions are caused by coding errors in the application program. The last four errors indicate a hardware failure or, in the case of "no memory available", a configuration problem. These are beyond control of the application program.

A procedure, ARMTRAP, is provided so that, if a trap occurs, control is returned to the application program. The application program is notified of the particular trap condition.

SECURITY

The GUARDIAN operating system's security capability is designed to fulfill four objectives:

- To prevent inadvertent destruction of files through purging or overwriting
- To prevent unauthorized access to sensitive data files by programmers or operations personnel
- To prevent unauthorized interference with running programs (processes)
- To provide a means of controlling intersystem accesses between network nodes

Security is enforced by assigning a group name, a user name, and (optionally) a password to individuals that are to access the system. File security may be set at three levels:

- User -- Only the user that created a file (a file's owner) may access the file.
- Group -- Only members of the group associated with the file's owner may access the file.
- Any -- Any user of the system may access the file.

For each file, file access at each level may be restricted to reading, writing, executing, and/or purging.

To provide control over system security, a system has a single user designated the "super ID". The super ID is responsible for creating new groups in the system. Each group has a single user that is designated the group manager; the group manager is responsible for creating new users in its group. The super ID additionally has full access to any file in the system.

```
?SOURCE $SYSTEM.SYSTEM.EXTDECS ( OPEN, READ, WRITE, CLOSE,
?                                NEWPROCESS, ABEND, STOP, MYTERM )
```

compiles only the external declarations for the OPEN, READ, WRITE, CLOSE, NEWPROCESS, ABEND, STOP, and MYTERM procedures.

SECTION 2

FILE SYSTEM

This section provides a general overview of the following:

- Files
- File Access
- Access Coordination among Multiple Accessors
- Wait/No-Wait I/O
- File System Implementation
- Error Indication
- Error Recovery

FILES

Input/output operations are performed by transmitting blocks of data between an application process and files. A file can be all or a portion of a disc, a device such as a terminal or a line printer, a process (i.e., running program), or the operator console.

A file is referenced by the symbolic file name that is assigned when the a file is created. A file name consists of two to twenty-four characters.

Disc Files

The ENSCRIBE (TM) Data Base Record Manager, an integral part of the GUARDIAN operating system, provides access to and operations on disc files. The ENSCRIBE software supports the following four file types:

- Key-Sequenced Files (records are placed in a file in ascending sequence according to the value of a "key field" in the record)
- Relative Files (records are stored relative to the beginning of the file)
- Entry-Sequenced Files (records are appended to a file in the order they are presented to the system)

used by other files.

Also specifiable at disc file creation is an optional "file code". This is an integer whose meaning is entirely application-dependent (except that codes 100 through 999 are reserved for use by Tandem Computers Inc.).

For a disc drive having a removable pack, the disc file can be designated at SYSGEN (system generation) time to have a "logically" removable volume. (A disc drive may, in fact, have a "physically" removable volume that will never be removed.) To mount a new volume in place of a currently mounted volume, the MOUNT command of the Peripheral Utility Program (PUP) is used. Logical interlocks exist in the file system to ensure that an in-use volume cannot be demounted (this interlock can be overridden) and that once the command is given to mount a new volume, further accesses to the mounted volume are prohibited.

Operations with disc files are described in detail in the ENSCRIBE Programming Manual.

Non-Disc Devices

Non-disc devices are items such as terminals (both conversational and page mode), line printers, magnetic tape units, card readers, and data communications lines. A file representing a non-disc device is referenced by a symbolic "device name" or a "logical device number". Device names and their corresponding logical device numbers are assigned at SYSGEN time.

What constitutes an input/output transfer with non-disc devices is dependent on the characteristics of the particular device involved. On a conversational mode terminal, for example, a transfer is one line of information; on a page mode terminal, a transfer is one page of information; on a line printer, a transfer is one line of print; on a magnetic tape unit, a transfer is one physical record on tape.

Operations with non-disc devices are described in detail in

- Section 2.5. TERMINALS
- Section 2.6. LINE PRINTERS
- Section 2.7. MAGNETIC TAPES
- Section 2.8. CARD READERS

Note: The ENVOY (TM) Data Communications Manager, an extension to the GUARDIAN operating system, provides an interface between application programs running in NonStop and NonStop II systems and data communications networks. Some features of ENVOY are:

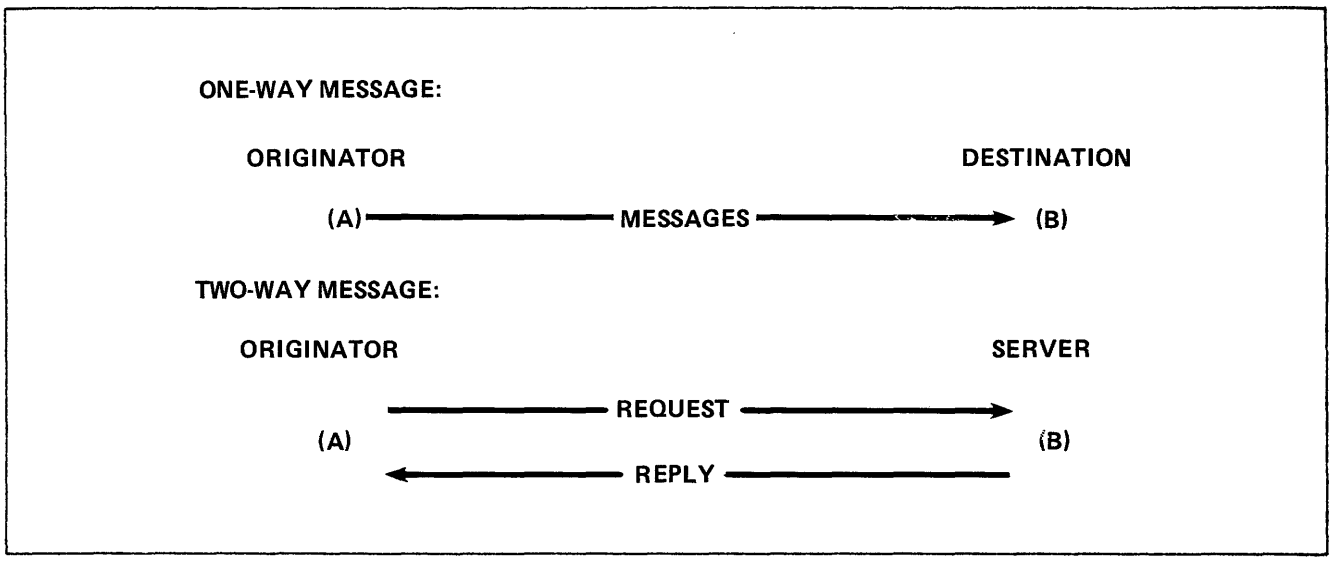


Figure 2-2. Communication with a Process via Process ID

The process name form of the process ID uniquely identifies a process or a process pair in the system. Process names can be predefined so that processes can be known throughout the system in the same manner as other device types (e.g., line printer) are known throughout the system. If a process [pair] is to be identified by the process name form of the process ID, its process name (which can be either application-defined or system generated) is assigned before the new process is created. A process name consists of a dollar sign "$" followed by one to five alphanumeric characters (the first must be alphabetic), optionally followed by one or two "qualification" names (see "File Names", section 2.2).

As shown in figure 2-3, there are certain NonStop aspects involved if communicating with a process pair. The primary process of the pair, while it is operable, receives (and replies to) all communications. If the primary process or its processor module fails, the backup process becomes the primary process and receives (and replies to) communications. The switch from the primary process to the backup process as the destination of a communication is performed automatically by the file system and is invisible to the originator of the message.

Several interprocess messages can be read and queued by the application process before a reply need be made. If one or more messages are to be queued, the maximum number of messages that the application process expects to queue must be specified. To identify each incoming message and direct a reply back to the originator of the message, a "message tag" must be obtained in a call to a file system procedure. When reply is sent for a particular message, the message's associated "message tag" is passed back to the system.

Interprocess communication is described in detail in

- Section 2.9. INTERPROCESS COMMUNICATION

Operator Console

A process may log messages on the operator console through a special file referenced by the file name $0 (verbally, "dollar zero"). The operator console is a write-only file (i.e., can be written to only). Console messages are prefixed with the current date and time and the ID of the process that logged the message. There is no special format imposed for logging messages on the operator console.

Operations with the operator console are described in detail in

- Section 2.10. OPERATOR CONSOLE

FILE ACCESS

Communication between an application process and a file is established through the file system OPEN procedure. An array in the application process's data area, containing the symbolic file name of the file to be accessed, is passed as a parameter to the OPEN procedure. In return, OPEN provides a process-unique "file number" that is used to identify the file when making subsequent file system procedure calls.

For example, to establish communication (open a file) with a terminal referenced by the device name "$TERM1", the following would be written in an application program:

```
INT .filename[0:11] := ["$TERM1",9 * ["  "]], ! data declarations.
    filenum,numxferred,                       !
    .buffer[0:35];                            !
```

Communication is established using the OPEN procedure:

```
    .
CALL OPEN(filename,filenum);
    .
```

OPEN establishes communication with the terminal identified by $TERM1. A process-unique file number is returned in "filenum".

```
CALL OPEN (disc^fname,filenum);
    .
```

opens a disc file referenced by the file name in "disc^fname".

Associated with each open disc file are three pointers: a "current-record" pointer, a "next-record" pointer, and an "end-of-file" pointer. Upon opening a file, the current-record and next-record pointers are set to point to the first byte in the file. A read or write operation always begins at the byte pointed to by the next-record pointer. The next-record pointer is advanced with each read or write operation by the number of bytes transferred; this provides automatic sequential access to a file. Following a read or write operation, the current-record pointer is set to point to the first byte affected by the operation. The next-record and current-record pointers can be set to an explicit byte address in a file, thereby providing random access. The end-of-file pointer contains the relative byte address of the last byte in a file plus one. The end-of-file pointer is automatically advanced by the number of bytes written when appending to the end of a file.

Sequential access to an unstructured disc file is implied. A data transfer operation with an unstructured disc file always starts at the location pointed to by the current setting of the next-record pointer:

```
    .
CALL READ(filenum,buffer,512,numxferred);
    .
```

transfers 512 bytes from the disc file starting at relative byte zero into "buffer". The next-record pointer is incremented by 512, the current-record pointer points to relative byte zero.

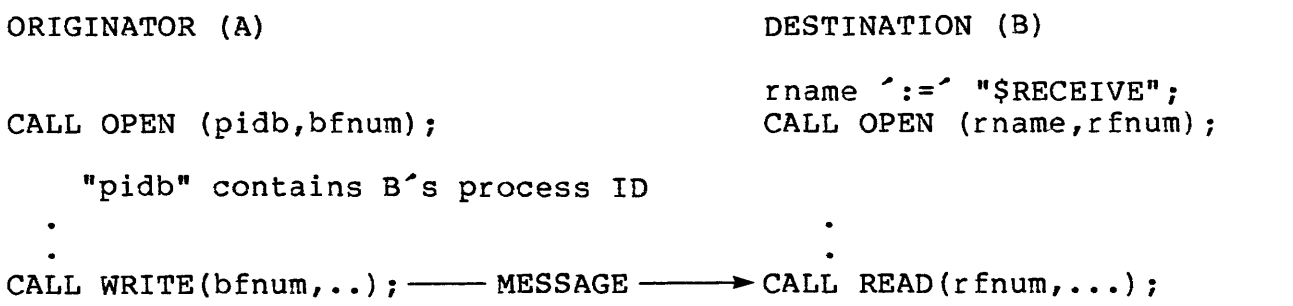
```
CALL READ(filenum,buffer,512,numxferred);
```

transfers 512 bytes from the disc file starting at file byte 512 into "buffer". The next-record pointer is incremented by 512 and now points to relative byte 1024; the current-record pointer points to relative byte 512.

Random access to a disc file is provided by the file system POSITION procedure. This procedure is used to set the current-record and next-record pointers:

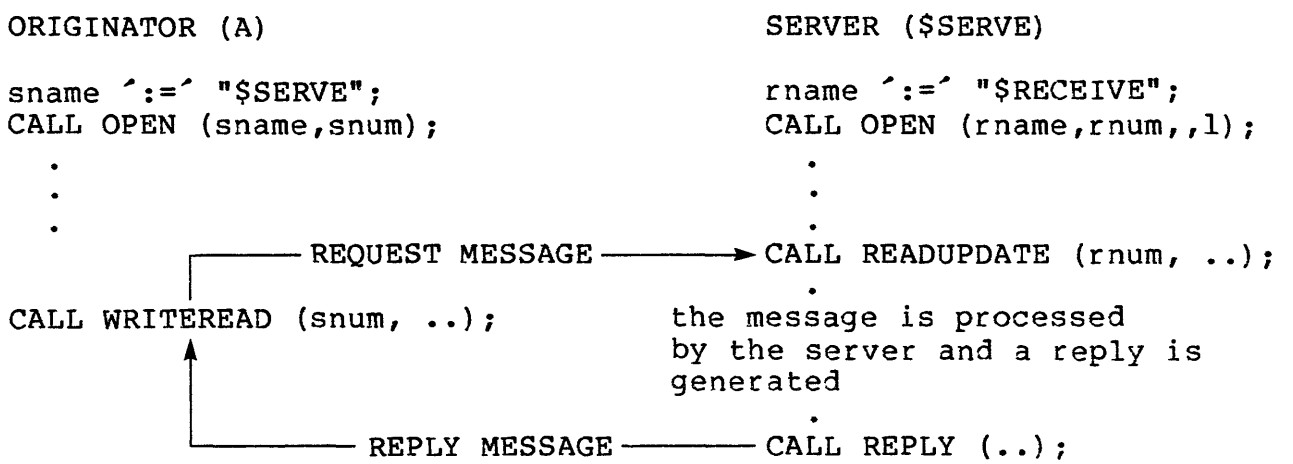
```
    .
CALL POSITION(filenum,4096D);
    .
```

positions the file pointers to point at relative byte 4,096.



A sends a message to B via B's process ID. B reads the message via its $RECEIVE file.

A two-way message could occur with a process identified by the process name form of process ID as follows:



A sends a request to $SERVE and waits for a reply in the call to WRITEREAD. $SERVE reads the message from its $RECEIVE file via a call to READUPDATE. When the reply is ready, it is sent back to A via a call to REPLY. When A receives the reply, WRITEREAD completes and A resumes processing.

ACCESS COORDINATION AMONG MULTIPLE ACCESSORS

A file may be accessed by several different processes at the same time. In order to coordinate simultaneous access, each process must indicate, when opening the file, how it intends to use the file. Both an access mode and an exclusion mode must be specified.

The "access mode" specifies the operations that will be performed by an accessor. The access mode is specified as one of the following:

- Read/Write (default access mode)
- Read-Only
- Write-Only

WAIT/NO-WAIT I/O

The file system provides the capability for an application process to execute concurrently with its file operations.

Two definitions:

- Wait I/O (the default)

  "Wait" i/o means that when designated file operations are performed (i.e., via file system calls), the application process is suspended, waiting for the operation to complete.

- No-wait I/O

  "No-wait" i/o means that when designated file operations are performed, the application process is not suspended. Rather, the application process executes concurrently with the file operation. The application process waits for an i/o completion in a separate file system call.

The operation of wait and no-wait i/o is illustrated in figure 2-5.

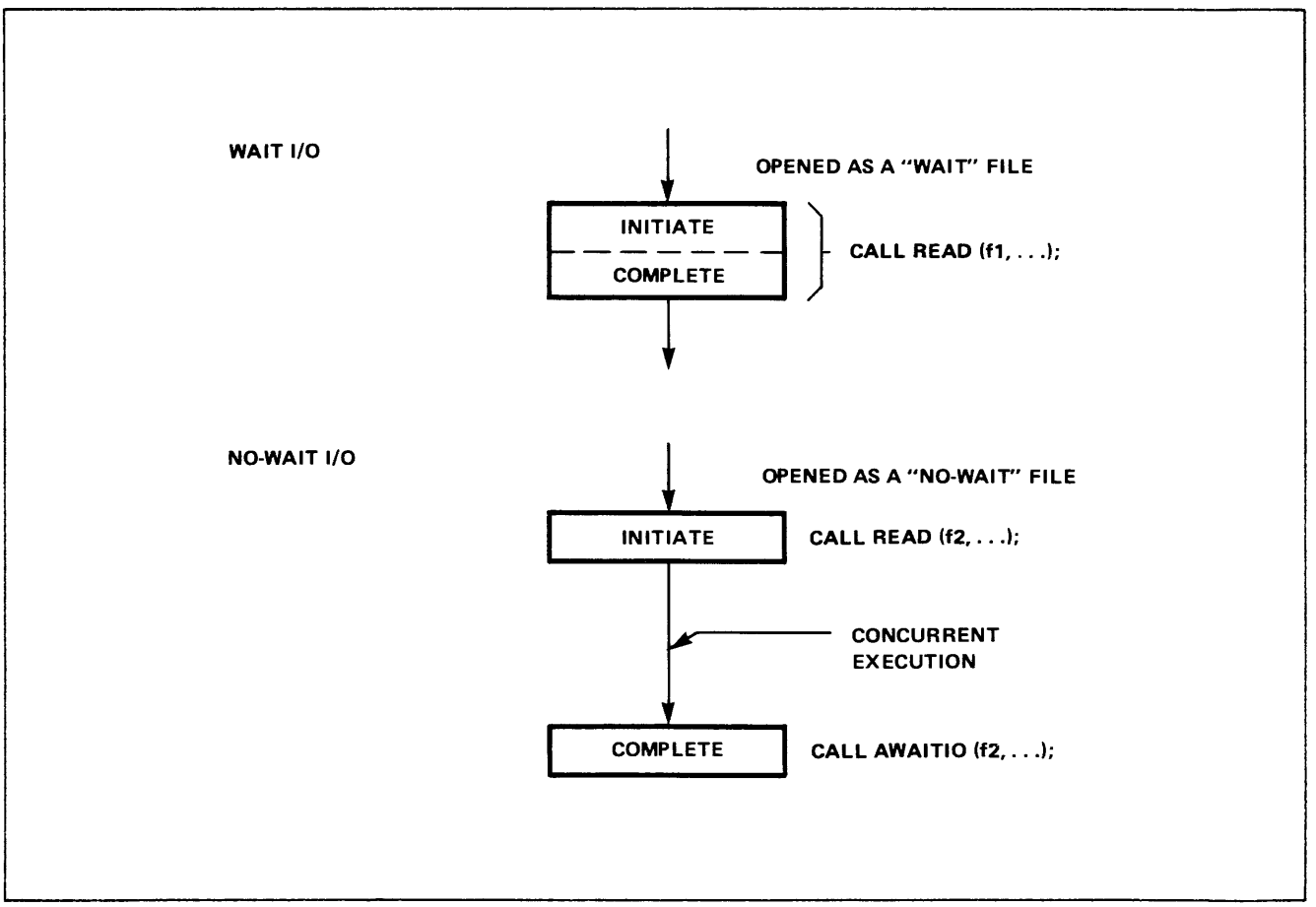


Figure 2-5. Wait versus No-Wait I/O

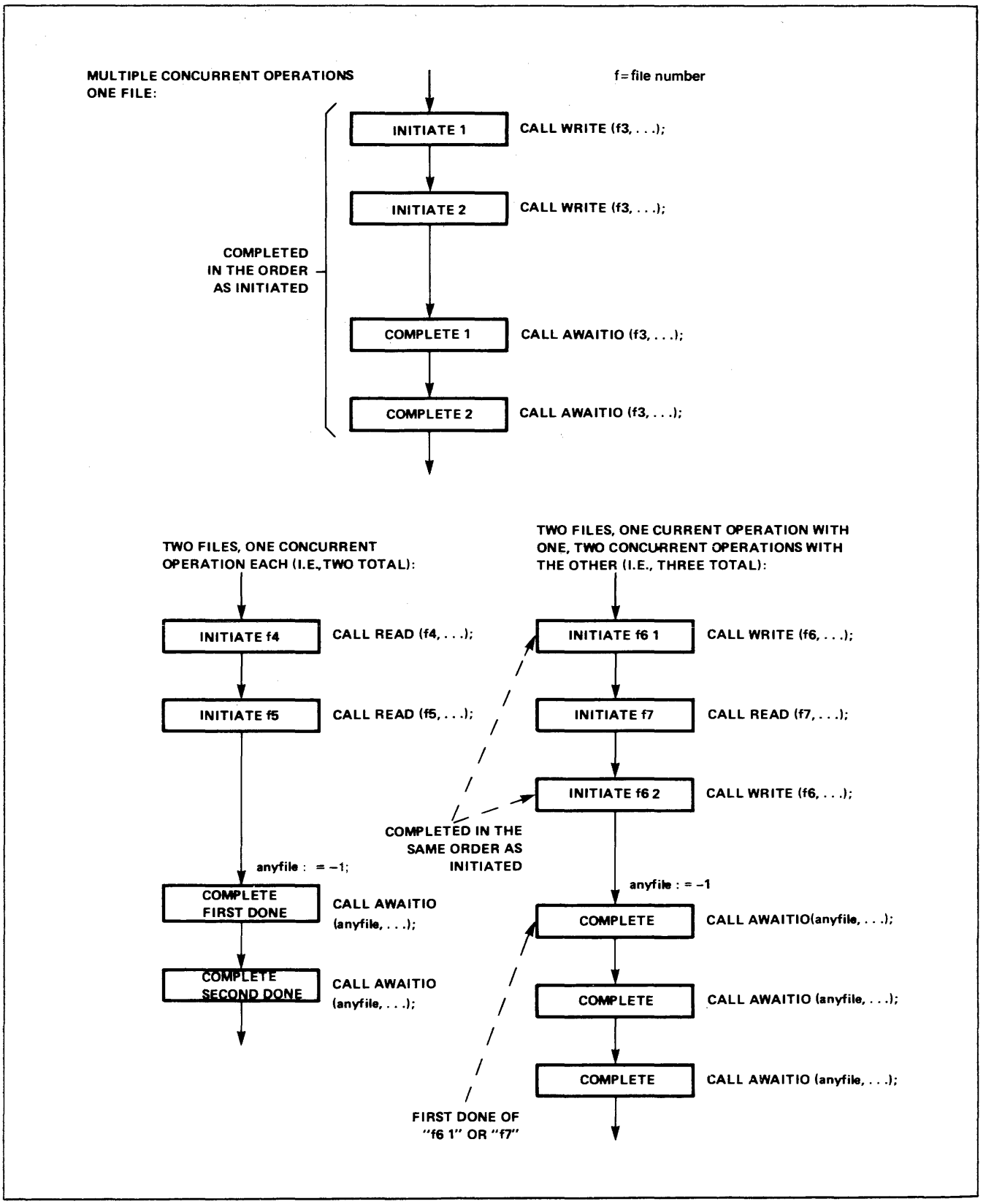


Figure 2-6. No-Wait I/O (Multiple Concurrent Operations)

connected to a single channel.)

- The i/o controller

  The i/o controller provides the electrical interface between an i/o device and the i/o channel. (I/O controllers are generally capable of controlling multiple devices.)

Two physically independent communication paths are accomplished as follows:

- The two interprocessor buses provide two independent communication paths between processor modules. If either bus fails, the other is still available.

- I/O controllers have two interface ports and are connected to the i/o channels of two processor modules. If one channel fails, control of the i/o controller is accomplished via the i/o channel connected to the other processor module.

The hardware i/o structure is depicted in figure 2-7.

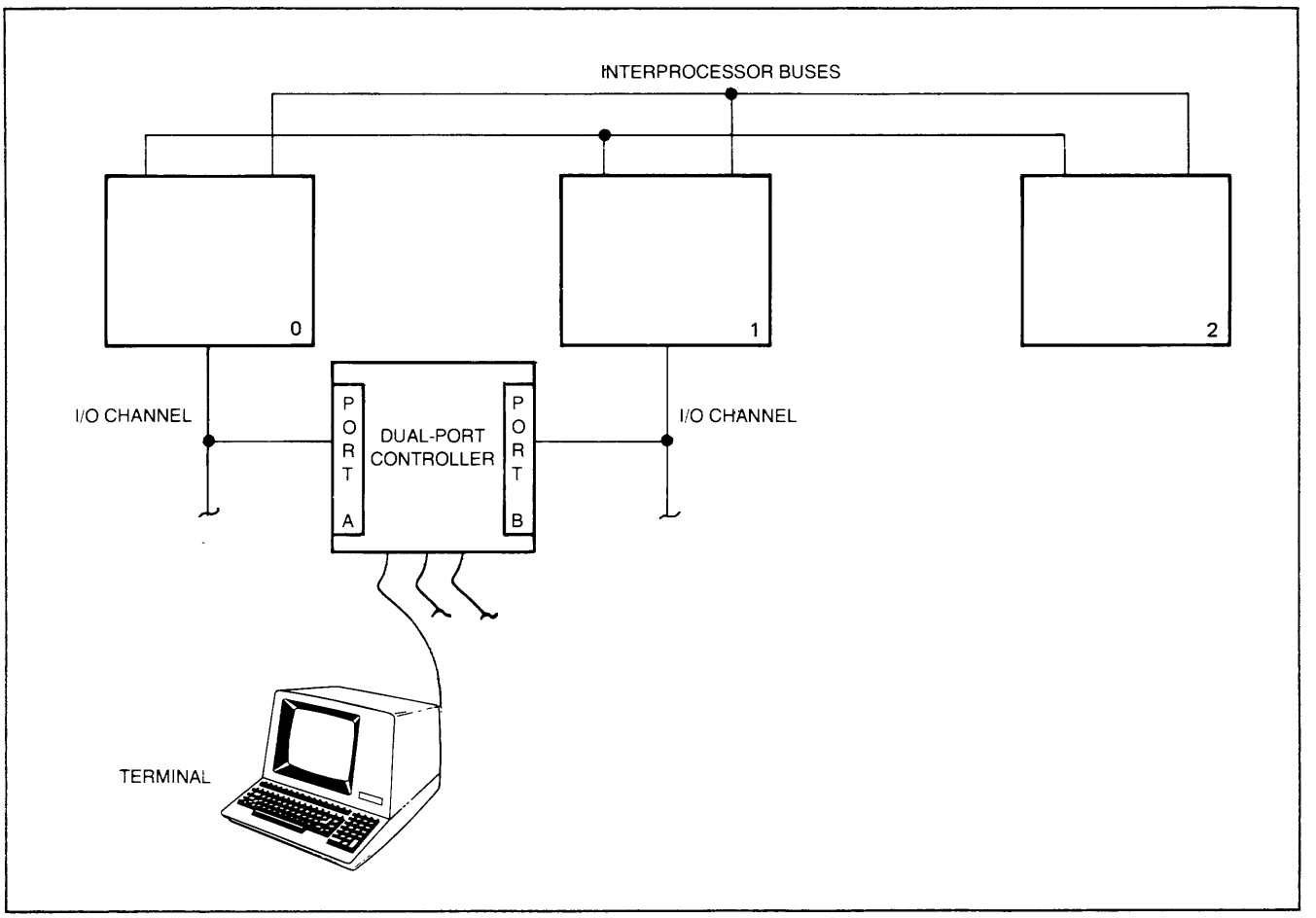


Figure 2-7. Hardware I/O Structure

original primary processor module is reloaded. (See the NonStop System Management Manual or the NonStop II System Management Manual for an explanation of "cold load" and "reload".)

Figure 2-8 depicts the primary and alternate communication paths to a device. While the primary path is operable, all i/o transfers occur via that path. Only when a failure of the primary path is detected does the alternate path come into use. Once an alternate path is brought into use, it becomes the primary path and is used exclusively.

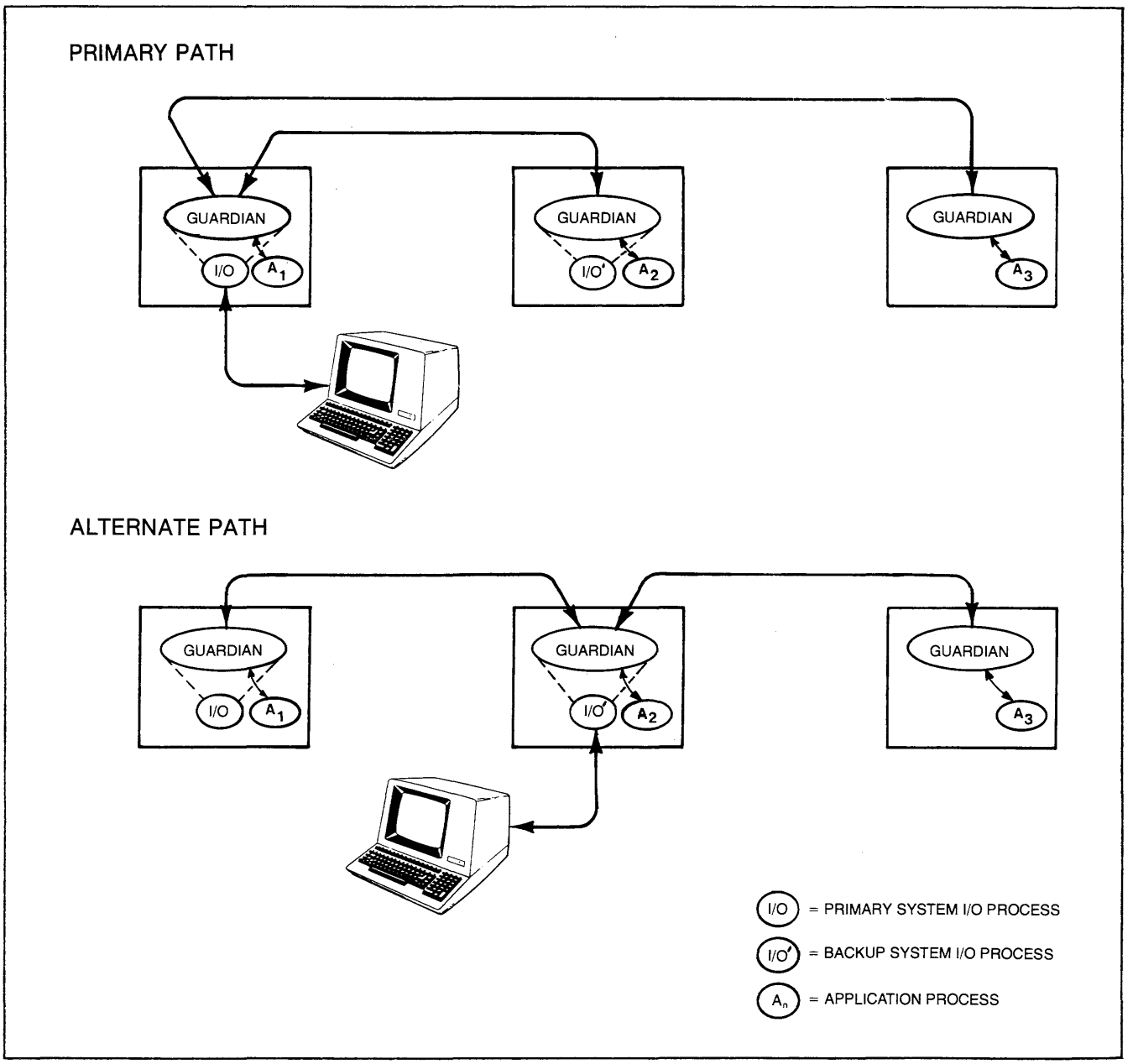


Figure 2-8. Primary and Alternate Communication Paths

File System Procedure Execution

File system procedures reside in operating system code, but execute in the application process's environment. When a file system procedure (or any operating system procedure, for that matter) is called by an application process, the system procedure's local storage is allocated in the application process's data stack, as shown in figure 2-9. The maximum amount of local storage required by a call to a system procedure is approximately 400 words.

File Open

The OPEN procedure establishes a communication path to a file. The symbolic file name that identifies a file is used to search a table, a copy of which resides in each processor module, called the Logical Device Table. The Logical Device Table contains an entry for each device connected to the system. Each entry contains a device name or, in the case of disc files, a volume name, the process ID of the primary system i/o process that controls the device/volume, and the process ID of the backup system i/o process that controls the device/volume.

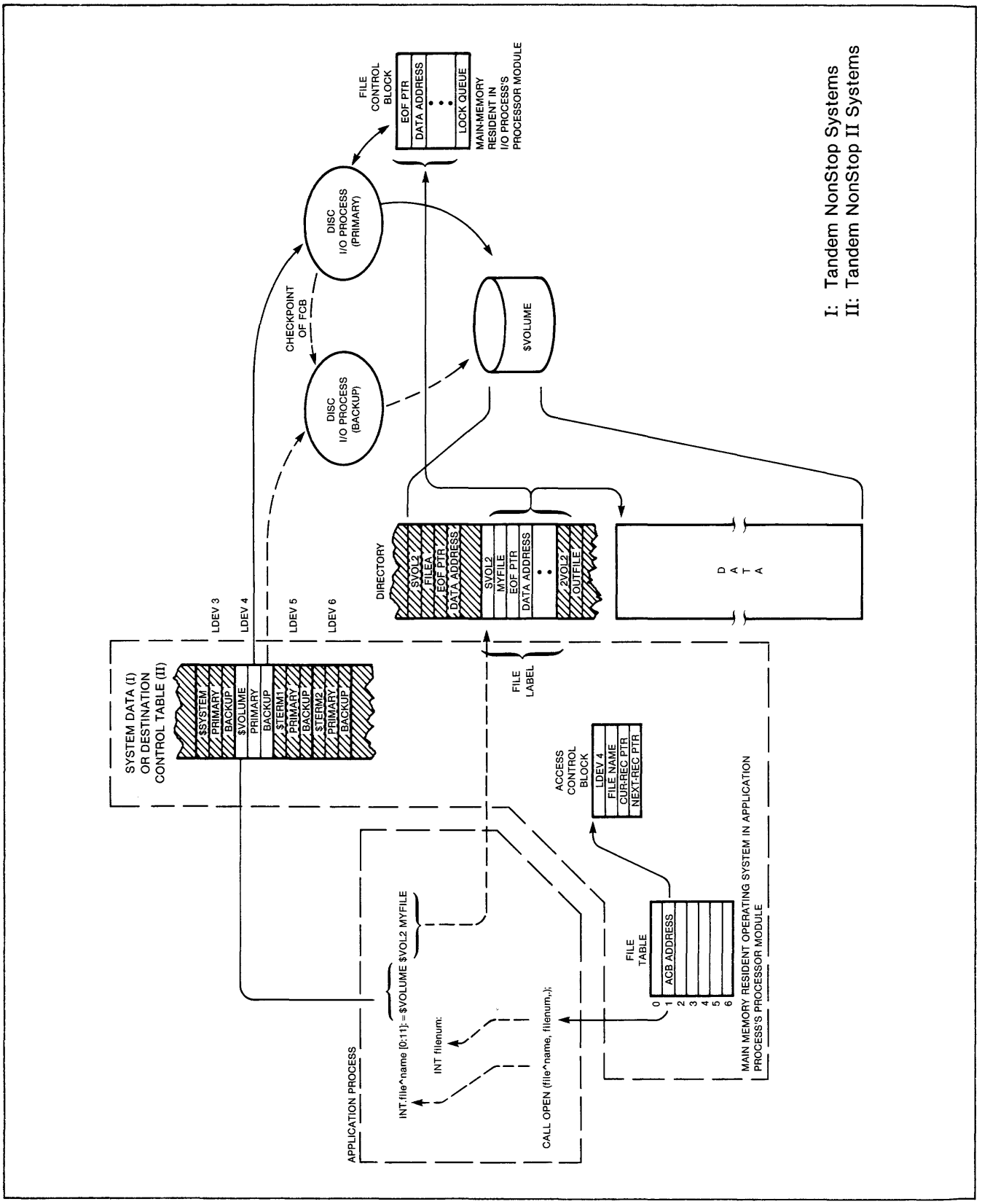


Figure 2-10. File Open

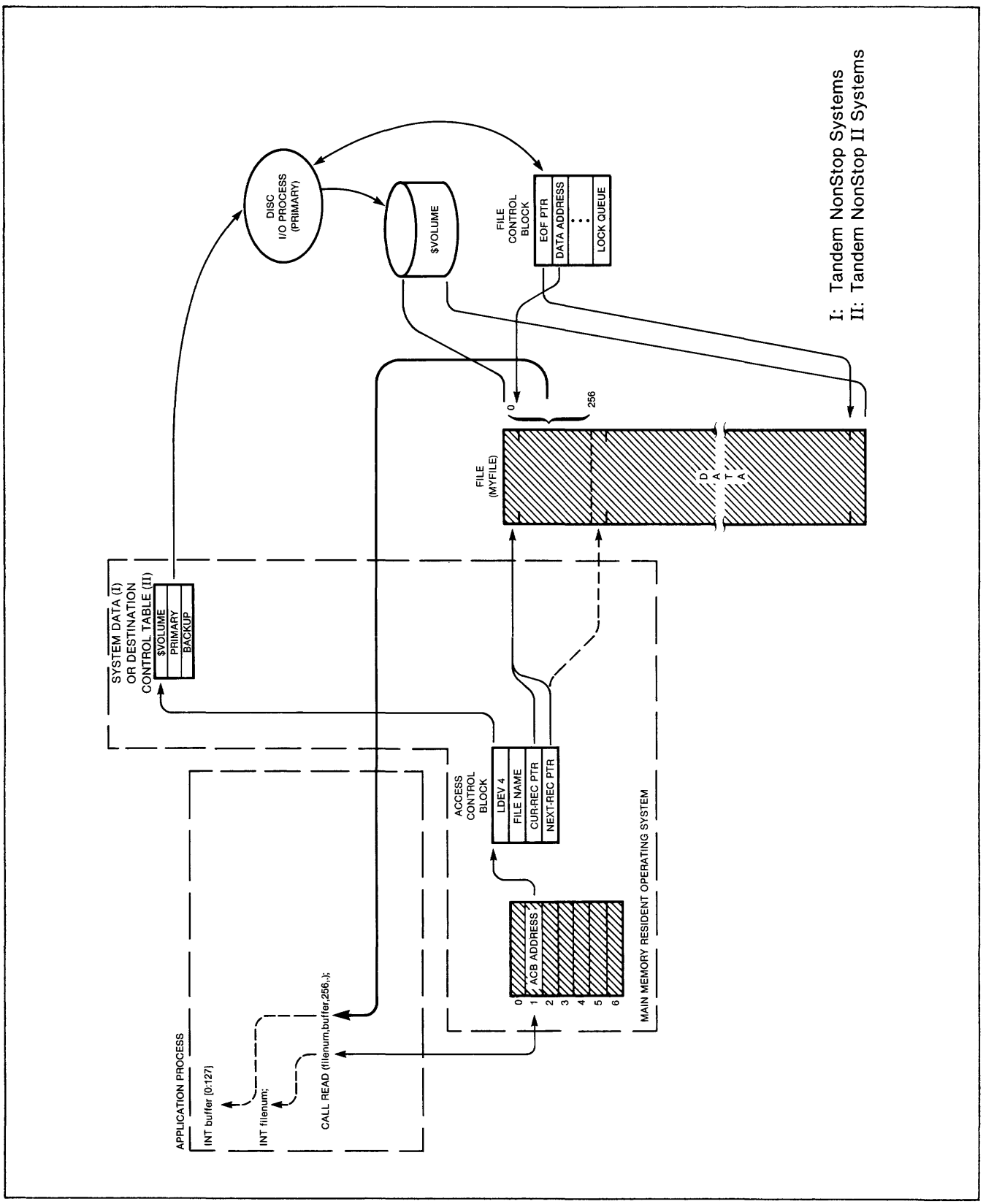


Figure 2-11. File Transfer

At this point, the file system (executing on behalf of the application process) moves the data from the resident File System buffer to an array in the application process's (virtual) data area.

On NonStop systems, File System Buffers are obtained from a memory space pool, called SHORTPOOL, in the operating system's data area. Processes requiring File System Buffers compete for this space on a first-come, first-served basis. If space is not available when needed, the application process is suspended until either the needed space becomes available or a configured timeout period expires; in the latter case, an error indication is returned to the application process. When an i/o transfer is completed, the space in use by the File System Buffer is returned to SHORTPOOL for use in subsequent data transfers.

On NonStop systems, there are three types of I/O Buffers (the type of buffer that a device uses is specified at system generation time):

- Pooled buffers - buffer space is secured from an i/o buffer pool, called IOPOOL, in the operating system's data area. I/O processes controlling devices using pooled buffers compete for space on a first-come, first-served basis. If space is not available when needed, the i/o process is suspended until either the needed space becomes available or a configured timeout expires; if a timeout occurs, an error indication is returned to the application process. When an i/o transfer is completed, the i/o buffer space is returned to IOPOOL for use in subsequent data transfers.

- Shared buffers - buffer space in the operating system data area is shared among two or more i/o devices on the same controller.

- Dedicated buffers - buffer space in the operating system data area is dedicated to a single device.

On NonStop II systems, File System Buffers are obtained from the process's Process File Segment (PFS). I/O Buffers are obtained from the i/o segments as needed by the i/o process. Processes that require dedicated buffers obtain buffer space during initialization. Once a process has obtained dedicated buffer space, it keeps that space until it terminates execution.

File Close

When a file is closed, the communication path to the file is broken. The Access Control Block is deleted, and the space that it used is returned for use as another Access Control Block. In the case of disc files, if no other opens are outstanding for the file, then the File Control Block is also released, and information such as the end-of-file pointer and addresses of allocated extents is updated on the physical disc from the information that was maintained in the File Control Block.

The backup i/o process, when notified of the primary's failure, takes over the primary's duties. The first action that the backup performs is to execute the i/o operation indicated by the latest checkpoint message received from the primary i/o process (this occurs regardless of whether the operation had been completed by the primary).

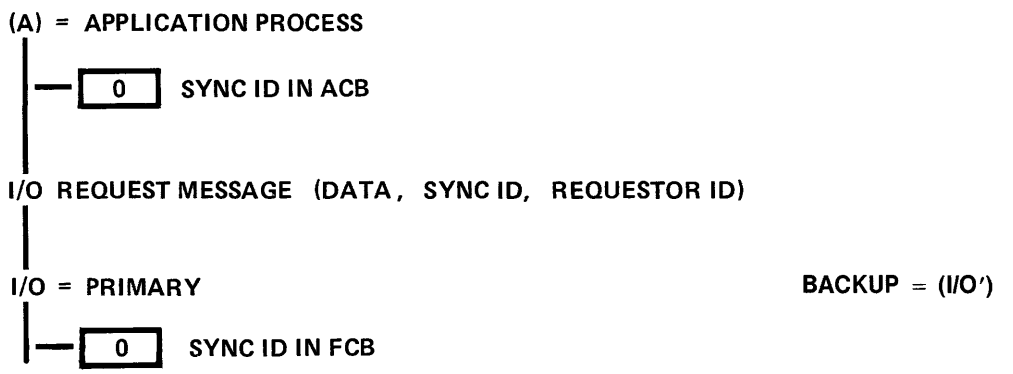
When the file system receives notification of the primary's processor module failure, after an operation has been requested but before it has been notified by the i/o process of a successful completion, it reinitiates the operation, this time sending the i/o request message (containing the data, sync ID, requestor ID, and disc address) to the backup i/o process.

Following a takeover from its primary, the backup i/o process checks the sync ID and requestor ID in the i/o request message for a match in the list of completed operations. If there is a match, the requested operation has already completed, and the backup i/o process returns the associated completion status to the file system; no other action is taken. If there is no match, the backup i/o process has not performed the operation. The operation is performed in its entirety, and the operation's completion status is returned to the file system.

The first operation is performed without incident:

```
CALL WRITE(fnum,...);
```

1. The file system sends an i/o request message to the primary disc i/o process.



2. In the primary i/o process:

* - The sector to be updated is read from disc.
- The sector image in memory is updated.
- The next sync ID (1) is saved.

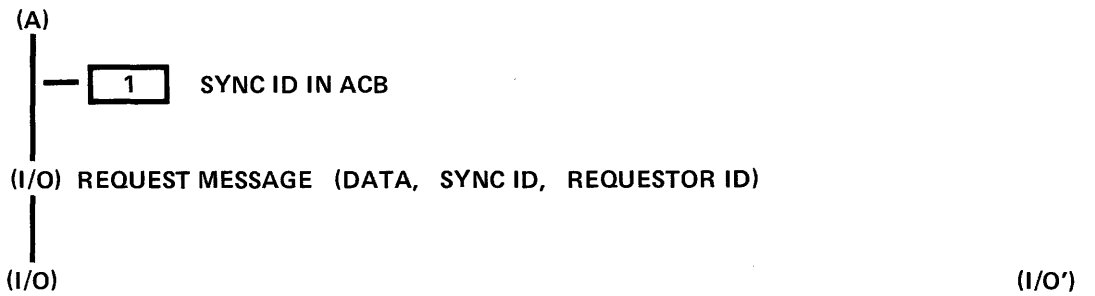
* performed only if partial-sector write

6. The file system increments the sync ID in the ACB.

The next operation encounters a failure:

```
CALL WRITE (fnum,...);
```

1. The file system sends an i/o request message to the primary disc i/o process.

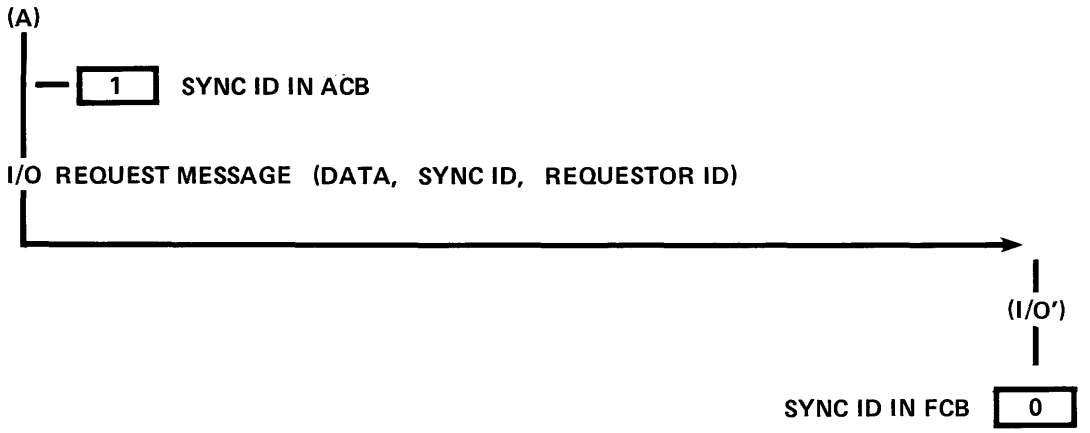


2. In the primary i/o process:

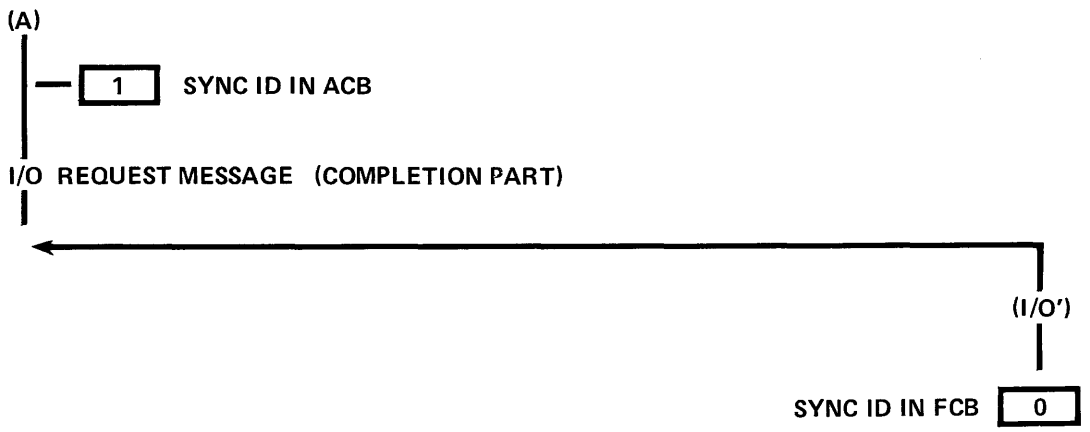
* - The sector to be updated is read from disc.
- The sector image in memory is updated.
- The next sync ID (0) is saved.

* performed only if partial-sector write

5. The file system, on behalf of the application process, reinitiates the request, this time to the backup process.



6. The backup i/o process compares the requestor ID and sync ID in the i/o request message with that of operations it has already performed. (*) The backup recognizes that this is a request to perform an operation it has already completed. Therefore, the operation is not performed. Rather, the completion status from the completed operation is returned to the file system.



7. The file system increments the sync ID in the ACB.

* performed only if partial-sector write

When a write is performed to a mirror volume, the (primary) i/o process automatically writes the data on the two disc devices comprising the volume. Both devices, when both are operable, are used by the i/o process for reading. If one of the devices becomes inoperable, the i/o process performs all subsequent reading from the operable device.

When an inoperable device is repaired, the information on the previously inoperable pack is brought up to date by means of the PUP (Peripheral Utility Program) "REVIVE" command. The REVIVE command copies the information from the operable pack onto the previously inoperable pack in groups of one or more tracks. This copying operation is carried out concurrently with requests to read or update data in files on this volume. (An optional parameter to the REVIVE command specifies a time interval between copying groups of tracks. This permits the revive operation to take place without a significant degradation of system performance.)

Four options are provided to optimize mirror volume performance when both devices of a mirror volume are operable. These options, which are specified at system generation time, are:

- for reading, SLAVESEEKS or SPLITSEEKS

  SLAVESEEKS specifies that both devices of a mirror volume are to seek (i.e., perform head positioning) together. The device that is to be used for reading data is selected at random.

  SPLITSEEKS specifies that the device with its head positioned closest to the desired cylinder is the device to be used for reading. The alternate device's head is not repositioned.

- for writing, SERIALWRITES or PARALLELWRITES
  (10 MB and 50 MB discs only; available only on NonStop systems)

  SERIALWRITES specifies that both devices are to seek together when preparing to write. The actual data transfer completes on one device before beginning for the other.

  PARALLELWRITES specifies that both devices are to seek together when preparing to write. Data transfers to both devices occur concurrently. This option is allowed only if each device is controlled by a separate hardware controller.

ERROR INDICATION

For all devices, each file system procedure sets the hardware condition code to indicate the outcome of an operation. The condition code settings have the following meanings:

<	(CCL)	indicates that an error occurred
=	(CCE)	indicates that the operation was successful
>	(CCG)	indicates a warning

returns, in "error", the error number associated with the last operation with the file represented by "filenum".

Specific errors are described in detail in the following sections of this manual:

- Section 2.4. FILE SYSTEM ERRORS AND ERROR RECOVERY
- Section 2.5. TERMINALS
- Section 2.6. LINE PRINTERS
- Section 2.7. MAGNETIC TAPES
- Section 2.8. CARD READERS
- Section 2.9. INTERPROCESS COMMUNICATION
- Section 2.10. OPERATOR CONSOLE

ERROR RECOVERY

In general, errors can be categorized as follows:

1. No error

2. Informational

3. Soft (recoverable)

4. Hard (not recoverable)

5. Path errors (recoverable)

The "informational" errors are those classified as "warnings". For example:

1	logical end-of-file encountered
6	system message received

The "soft" errors are those for which programmatic recovery is possible or the error condition can be expected to go away. These include errors such as

10	file already exists
11	file not in directory
40	operation timed out
73	file locked
100	device not ready
101	no write ring (magnetic tape)
102	paper out (line printer)
110	only BREAK request allowed to terminal
111	terminal operation aborted because BREAK key typed

Errors 100 - 102 require operator intervention to correct the error condition.

The "hard" errors are those for which programmatic recovery is not possible. These include

File names are used to access devices, disc files, processes, and the operator console through the file system OPEN procedure. Additionally, file names are used when creating new disc files, purging old disc files, and renaming disc files.

There are two forms of file name - external and internal. The "external" form is used when entering file names into the system from the outside world (e.g., by a user to specify a file name to the Command Interpreter). The external form is described in section 11, "COMINT/Application Interface". The "internal" form is used within the system when passing file names between application processes and the operating system. This section describes the internal form (see the EXPAND Users Manual for the internal form of the network file names).

The conversion from external to internal form is performed automatically by the Command Interpreter for the IN and OUT file parameters of the RUN command (see section 11). For general conversion of file names from the external to the internal form, the FNAMEEXPAND procedure is provided.

The internal form of file names is:

```
<file name> ! 12 words, blank filled.

where

   to access permanent disc files, use

      <file name>[0:3]  =  $<volume name><blank fill>
      <file name>[4:7]  =  <subvol name><blank fill>
      <file name>[8:11] =  <disc file name><blank fill>

   to access temporary disc files, use

      <file name>[0:3]  =  $<volume name><blank fill>
      <file name>[4:11] =  the <temporary file name> returned by
                           CREATE (which is blank filled)

   to access non-disc devices, use

      <file name>[0:11] =  $<device name><blank fill> or
                           $<logical device number><blank fill>

   to communicate with other processes, use

      <file name>[0:11] =  $RECEIVE<blank fill>

         to perform READ, READUPDATE, and REPLY operations, and
                                                              →
```

Examples

Permanent disc file:

```
INT .fname[0:11] := "$STORE1 ACCT1   MYFILE  ";
```

Temporary disc file:

```
INT .fname[0:11] := ["$STORE1 ", 8 * ["  "]];
```

only the volume name is supplied. The temporary file name is returned from CREATE.

```
CALL CREATE(fname);
```

DEVICE NAMES

Device names identify particular input/output devices in the system. They are assigned to the logical devices at system generation time. A device name must be preceded by a dollar sign "$" and consists of a maximum of seven alphanumeric characters; the first character must be alphabetical.

Example:

```
INT .fname[0:11] := ["$TERM1", 9 * ["  "]];
```

LOGICAL DEVICE NUMBERS

Logical device numbers identify entries in the logical device table which, in turn, identify particular input/output devices in the system. Logical device numbers are assigned to physical i/o devices when system generation occurs (SYSGEN). A logical device number must be preceded by a dollar sign "$" and consists of a maximum of four numerical characters; the maximum logical device number is 2047.

A process can determine the logical device number of its home terminal by calling the MYTERM utility procedure.

Example:

```
INT .fname[0:11] := ["$0012 ", 9 * ["  "]];
```

$RECEIVE

$RECEIVE is a special file name used to receive and reply to messages from other processes.

Example:

```
INT .fname[0:11] := ["$RECEIVE", 8 * ["  "]];
```

If a process name represents a process pair and the process accessing the pair is a member of the pair, then the process name references the opposite member of the pair.

OPTIONAL QUALIFICATION OF PROCESS NAMES. The process name form of a process ID can be further qualified at file open time by the addition of one or two optional "qualifier" names. This provides for process file names of the form:

```
word:
[0:3]                [4:7]                    [8:11]
$<process name>   [ #<1st qualif name> [ <2nd qualif name> ] ]
```

where

#<1st qualif name>

consists of a number sign "#" followed by one to seven alphanumeric characters, the first of which must be alphabetical.

<2nd qualif name>

consists of one to eight alphanumeric characters, the first of which must be alphabetical.

Note that only the process name has meaning to the file system (it indicates the particular process [pair] being opened). The qualifier names have no particular meaning to the file system (they are, however, checked for being of the proper format). Instead, their meaning must be interpreted by the process being opened (these names are passed to the process being opened in an "OPEN" system message).

Obtaining a Process ID

A process ID can be obtained from a number of sources:

- When the process control NEWPROCESS procedure is called to create a new process, the process ID of the newly created process is returned. If a process name was also entered into the PPD in the call to NEWPROCESS, the process ID returned consists of

  ```
  <process designator>[0:2] = $<process name>
  <process designator>[3]   = <cpu,pin>
  ```

- A process can obtain the process ID of its creator by calling the process control MOM procedure. If a process's creator is in the PPD, the information returned is in the same form as that described above for the NEWPROCESS procedure.

NETWORK FILE NAMES

File names can optionally include a system number that identifies a file as belonging to a particular system on a network. (See the EXPAND Users Manual for information regarding networks of Tandem systems.)

In this context, a file name beginning with a dollar sign, "$", is said to be in "local" form, to distinguish it from a file name beginning with a backslash, "\", which characterizes the "network" form. Specifically, the network form of a file name is:

```
<network file name>  ! 12 words, blank filled

   word[0].<0:7>   =   \  (ASCII backslash)
   word[0].<8:15>  =   <system number>, in octal
   word[1:3]       =   <volume name>, <device name>, or
                       <process id>
   word[4:11]      =   same as local file name

where

   <system number>

      is an integer between 0 and 254 that designates a
      particular system.  The assignment of system numbers is
      made at system generation (SYSGEN) time.

   <volume name>

      consists of at most six alphanumeric characters, the first
      of which must be alphabetic.

   <device name>

      consists of at most six alphanumeric characters, the first
      of which must be alphabetic.

   <process id>

      is in either the timestamp form or the process name
      form, both of which are described below.
```

Note that names of disc volumes and other devices, when embedded within a network file name, are limited to having six characters, and do NOT begin with a dollar sign. Similar restrictions apply to the network form of the process ID, as follows.

The file system procedures are:

AWAITIO	waits for completion of an outstanding i/o operation pending on an open file
CANCELREQ	cancels the oldest outstanding operation, optionally identified by a tag, on an open file.
CLOSE	stops access to an open file and purges a temporary disc file
CONTROL	executes device-dependent operations on an open file
CONTROLBUF	executes buffered device-dependent operations on an open file
CREATE	creates a new disc file (permanent or temporary)
DEVICEINFO	provides the device type and physical record size for a file (open or closed)
EDITREAD EDITREADINIT	read text records from an edit format file
FILEERROR	is used to decide if an i/o operation should be retried
FILEINFO	provides error information and characteristics about an open file
FNAMECOLLAPSE	collapses an internal file name to its external form
FNAMECOMPARE	compares two internal format file names within a local or network environment
FNAMEEXPAND	expands a partial file name from the compacted form to the standard twelve-word internal form usable by the file system procedures
GETDEVNAME	returns the $<device name> or $<volume name> associated with a logical device number if such a device exists. Otherwise the name of the next higher logical device is returned
GETSYSTEMNAME	supplies the system name corresponding to a system number
LASTRECEIVE	provides the process ID and, optionally, the message tag associated with the last message taken from the $RECEIVE file
LOCATESYSTEM	provides the system number corresponding to a system name

SAVEPOSITION	is used to save disc file positioning information so that a return to that position can be made in a subsequent call to REPOSITION
SETMODE	sets device-dependent functions in an open file
SETMODENOWAIT	sets device-dependent functions in a no-wait manner for an open file.
UNLOCKFILE	unlocks an open disc file currently locked by the caller
WRITE	writes information to an open file
WRITEREAD	writes, then immediately reads back from an open terminal or data communications file For interprocess communication, WRITEREAD is used to originate a message to a designated process then wait for a reply message back from that process
WRITEUPDATE	is used for open disc files to update data in the location read by the last call to READ or READUPDATE (i.e., the position indicated by the setting of the current record pointer) For magnetic tapes, WRITEUPDATE is used to replace an existing record on tape (except on 5106 Tri-Density Tape Drive)

CHARACTERISTICS

For Procedure Usage by Device Type

ALL DEVICE TYPES. The following basic set of procedures apply to all device types:

DEVICEINFO, GETDEVNAME, OPEN, READ, WRITE, AWAITIO, CANCELREQ, SETMODE, SETMODENOWAIT, CONTROL, CONTROLBUF, FILEINFO, and CLOSE

UNSTRUCTURED DISC FILES. In addition to the basic set of procedures, the following procedures are used with unstructured disc files:

CREATE, NEXTFILENAME, POSITION, READUPDATE, WRITEUPDATE, LOCKFILE, UNLOCKFILE, RENAME, PURGE, REFRESH, SAVEPOSITION, and REPOSITION

<tag> Parameters

An application-specified double integer - INT(32) - tag can be passed as a calling parameter when initiating an i/o operation (e.g., read or write) with a no-wait file. The tag is passed back to the application process, through the AWAITIO procedure, when the i/o operation completes. The tag is useful for identifying individual file operations and can be used in application-dependent error recovery routines.

<buffer> Parameter

The data buffers in an application program used to transfer data between the application process and the file system must be integer (INT) or double integer (INT(32)) and must reside in the program's data area ('P' relative read-only arrays are not permitted).

<transfer count> Parameter

The transfer count parameter of file system procedures always refers to the number of BYTES to be transferred. The number of bytes that can be transferred in a single operation is dependent on the device involved:

device type	transfer count range	
disc	{0:4096}	
terminal	{0:4095}	(NonStop systems)
	{0:32767}	(NonStop II systems)
line printer	{0:4095}	(NonStop systems)
	{0:32767}	(NonStop II systems)
magnetic tape	{0:4095}	(NonStop systems, 3201 Controller)
	{0:4096}	(NonStop systems, 3202 Controller)
	{0:32767}	(NonStop II systems)
interprocess	{0:32000}	
operator console	{0:102}	

The above figures are file system/hardware maximums for the indicated devices. The actual maximum transfer count for a given device may be less than the above due to the physical characteristics of a particular device and/or the amount of buffer space assigned to the device at system generation time (SYSGEN).

For devices permitting odd count transfers, such as terminals and magnetic tapes, the value of the byte following the last byte of an odd count read is not meaningful.

The count of bytes is rounded up to an even number for transfers with unstructured disc files.

The AWAITIO procedure is used to complete a previously initiated no-wait i/o operation. AWAITIO can be used to:

- Wait for a completion with a particular file. Application process execution is suspended until the completion occurs. A timeout is considered to be a completion in this case.
- Wait for a completion with any file, or for a timeout to occur. A timeout is not considered to be completion in this case.
- Check for a completion with a particular file. The call to AWAITIO immediately returns to the application process regardless of whether there is a completion or not. (If there is no completion, an error indication is returned.)
- Check for a completion with any file.

If AWAITIO is used to wait for a completion, a time limit can be specified as to maximum time allotted to completing the waited-for operation.

The call to the AWAITIO procedure is:

```
CALL AWAITIO ( <file number>
---- ------- - -------------
               , <buffer address>
               , <count transferred>
               , <tag>
               , <time limit> )
                              -
```

where

<file number>, INT:ref:1,

if a particular file number is passed, AWAITIO applies to that file. The specific action depends on the value of the <time limit> parameter. If <time limit> is a nonzero value, the application process is suspended until a completion occurs or the time limit expires. If passed as 0D, a completion check is made.

if passed as -1, the call to AWAITIO applies to the oldest outstanding operation pending on any file. The specific action depends on the value of the <time limit> parameter. If <time limit> is a nonzero value, the application process is suspended until a completion occurs or the time limit expires. If passed as 0D, a completion check is made. In either case, if an operation completed, AWAITIO returns to <file number> the file number associated with the completion.

→

CONSIDERATIONS

- Normally, the oldest outstanding i/o operation is always completed first; therefore AWAITIO completes i/o operations associated with the particular open of a file in the same order as initiated.

  Specifying SETMODE 30 allows no-wait i/o operations to complete in any order. When initiating an i/o operation, an application process employing this option can use the <tag> parameter to keep track of multiple operations associated with an open of a file.

  Note: if SETMODE 30 is used, no-wait operations do not necessarily complete in the order they are returned by the i/o process, or in any other implied order.

- If an error indication is returned (i.e., condition code is CCL or CCG), the file number that is returned by AWAITIO can be passed to the FILEINFO procedure to determine the cause of the error. If <file number> = -1 (i.e., any file) is passed to AWAITIO and an error occurs on a particular file, AWAITIO returns, in <file number>, the actual file number associated with the error.

- Each no-wait operation initiated must be completed with a corresponding call to AWAITIO.

  - If AWAITIO is used to wait for completion (i.e., <time limit> <> 0D) and a particular file is specified (i.e., <file number> <> -1), then completing AWAITIO for any reason is considered a completion.

  - If AWAITIO is used to check for completion (<time limit> = 0D) or used to wait on any file (<file number> = - 1), completing AWAITIO does not necessarily indicate a completion. If an error indication is returned and a subsequent call to FILEINFO returns error 40 (i.e., a timeout), then the operation is considered incomplete (AWAITIO must be called again). Any indication other than error 40 (i.e., CCE, CCG, CCL and <error> <> 40) indicates a completion.

- If AWAITIO is called and a corresponding no-wait operation has not been initiated, an error indication is returned (CCL) and a subsequent call to FILEINFO returns error 26 (no outstanding operation).

- The contents of a buffer being written should not be altered between a no-wait i/o initiation (e.g., call to WRITE) and the corresponding no-wait i/o completion (i.e., call to AWAITIO). If the buffer is altered, application error recovery can become difficult, if not impossible. In addition, some programs which alter the buffer before AWAITIO completion may operate correctly on NonStop systems but fail on NonStop II systems.

The action of the AWAITIO procedure is illustrated in figure 2-14.

The CANCEL procedure is used to cancel the oldest outstanding operation on a no-wait file.

The call to the CANCEL procedure is:

```
CALL CANCEL ( <file number> )
---- ------ - ------------ -

where

   <file number>, INT:value,

     identifies the file whose oldest outstanding operation is to
     be canceled.

condition code settings:

   <  (CCL) indicates that an error occurred (call FILEINFO).
   =  (CCE) indicates that the operation was cancelled.
   >  (CCG) is not returned by CANCEL.

example:

   CALL CANCEL ( some^file );
   IF < THEN ..... ! no operation outstanding.
```

CONSIDERATIONS

- The function of CANCEL is a subset of those functions provided by CANCELREQ.

The CLOSE procedure is used to terminate access to an open file.

When a permanent disc file is closed, if it is not open concurrently, the file label on disc is updated with pertinent information from the main-memory resident File Control Block, and the space in use by the FCB is returned to a system main-memory space pool. When a temporary disc file is closed, if it is not open concurrently, its name is deleted from the volume's directory, and any space that had been allocated to the file is made available for other files.

For any file close, the space allocated to the Access Control Block is returned to the system.

The call to the CLOSE procedure is:

```
CALL CLOSE ( <file number> , <tape disposition> )
---- ----- - -------------                       -

where

   <file number>, INT:value,

     identifies the file to be closed.

   <tape disposition>, INT:value,

     specifies mag tape disposition:

     where

       <tape disposition>.<13:15>

         0 = rewind and unload, don't wait for completion
         1 = rewind, take offline, don't wait for completion
         2 = rewind, leave online, don't wait for completion
         3 = rewind, leave online, wait for completion
         4 = don't rewind, leave online

condition code settings:

   <  (CCL) indicates that the file was not open.
   =  (CCE) indicates that the CLOSE was successful.
   >  (CCG) is not returned by CLOSE.

example:

   CALL CLOSE ( tape^file, 1 );
```

The CONTROL procedure is used to perform device-dependent i/o operations.

If the CONTROL procedure is being used to initiate an operation with a file opened with no-wait i/o specified, the operation must be completed with a corresponding call to the AWAITIO procedure.

The call to the CONTROL procedure is:

```
CALL CONTROL ( <file number> , <operation> , <parameter>
               , <tag>                                    )
```

where

<file number>, INT:value,

identifies the file that is to execute the CONTROL operation.

<operation>, INT:value,

is defined by device in table 2-1.

<parameter>, INT:value,

is also defined in table 2-1.

<tag>, INT(32):value,

for no-wait i/o only, if present, is stored by the system, then passed back to the application process by the AWAITIO procedure when the CONTROL operation completes.

condition code settings:

<	(CCL)	indicates that an error occurred (call FILEINFO).
=	(CCE)	indicates that the CONTROL was successful.
>	(CCG)	for magnetic tape, indicates that the end-of-file was encountered while spacing records; for a process file, indicates that the process is not accepting CONTROL system messages.

example:

```
CALL CONTROL ( printer, form^control, vfu^channel );
IF < THEN ....                    ! error occurred.
```

Table 2-1. CONTROL Operations

Note: This table gives only a partial list of CONTROL operations, namely, those used with the i/o devices discussed in this manual. CONTROL operations used with other Tandem software products, such as ENVOY and AXCESS, are described in the manuals for those products.

<operation>

1 = terminal/line printer forms control

<parameter> for terminal or line printer (printer subtype 3)

0	= form feed (send %014)
1 - 15	= vertical tab (send %013)
16 or greater	= skip <parameter> - 16 lines

<parameter> for line printer (subtype 0 or 2)

0	= skip to VFU channel 0	(top-of-form)
1 - 15	= skip to VFU channel 1	(single space)
16 - 79	= skip <parameter> - 16 lines	

<parameter> for line printer (subtype 1 or 5)

0	= skip to VFU channel 0	(top-of-form)
1	= skip to VFU channel 1	(bottom-of-form)
2	= skip to VFU channel 2	(single space, top-of-form eject)
3	= skip to VFU channel 3	(next odd-numbered line)
4	= skip to VFU channel 4	(next third line: 1, 4, 7, 10, etc.)
5	= skip to VFU channel 5	(next one-half page)
6	= skip to VFU channel 6	(next one-fourth page)
7	= skip to VFU channel 7	(next one-sixth page)
8	= skip to VFU channel 8	(user-defined)
9	= skip to VFU channel 9	(user-defined)
10	= skip to VFU channel 10	(user-defined)
11	= skip to VFU channel 11	(user-defined)
16 - 31	= skip <parameter> - 16 lines	

<parameter> for line printer (subtype 4) (default DAVFU)

0	= skip to VFU channel 0	(top of form/line 1)
1	= skip to VFU channel 1	(bottom of form/line 60)
2	= skip to VFU channel 2	(single space/lines 1-60, top-of-form eject)
3	= skip to VFU channel 3	(next odd-numbered line)
4	= skip to VFU channel 4	(next third line: 1, 4, 7, 10, etc.)

Table 2-1. CONTROL Operations (cont'd)

10 = mag tape, space backward records

<parameter> = number of records {0:255}

11 = terminal or line printer (subtype 3 or 4), wait for modem connect

<parameter> = none

12 = terminal or line printer (subtype 3 or 4), disconnect the modem (i.e., hang up)

<parameter> = none

20 = disc, purge data (write access is required)

<parameter> = none

21 = disc, allocate/deallocate extents (write access is required)

<parameter> = 0 = deallocate all extents past the end-of-file extent
1:16 = number of extents to allocate

Note: A write end-of-file to an unstructured disc file sets the end-of-file pointer to the relative byte address indicated by the setting of the next-record pointer, and writes the new end-of-file setting in the file label on disc.

condition code settings:

<	(CCL)	indicates that an error occurred (call FILEINFO).
=	(CCE)	indicates that the CONTROLBUF was successful.
>	(CCG)	for a process file, indicates that the process is not accepting CONTROLBUF system messages.

example:

```
CALL CONTROLBUF ( printer, load^vfu, vfu^buffer, 132 );
IF < THEN ...                          ! error occurred
```

CONSIDERATIONS

- If a "wait" CONTROLBUF is executed, the <count transferred> parameter indicates the number of bytes actually transferred.
- If a "no-wait" CONTROLBUF is executed, <count transferred> has no meaning and can be omitted. The count of the number of bytes transferred is obtained when the i/o completes via the <count transferred> parameter of the AWAITIO procedure.

CONSIDERATIONS FOR INTERPROCESS COMMUNICATION

- The issuance of a CONTROLBUF to a file representing another process causes a CONTROLBUF system message (i.e., system message -35) to be sent to that process. If the object of the CONTROLBUF operation is not accepting CONTROL/CONTROLBUF/SETMODE system messages, the call to CONTROLBUF completes with a condition code of CCG; a subsequent call to FILEINFO returns error 7 (process not accepting CONTROL/CONTROLBUF/SETMODE messages).
- Any value may be specified for the <operation> parameter, and any data may be included in <buffer>. An application-defined protocol should be established for interpreting non-standard parameter values.
- CONTROLBUF is not valid for the $RECEIVE file.

The CREATE procedure is used to define a new disc file. The file can be either temporary (and therefore automatically deleted when closed) or permanent. If a temporary file is created, CREATE returns a file name suitable for passing to the OPEN procedure.

To create a structured disc file, refer to the ENSCRIBE Programming Manual. The call to the CREATE procedure for unstructured files is:

```
CALL CREATE ( <file name>
---- ------ - -----------
              , <primary extent size>
              , <file code>
              , <secondary extent size>
              , <file type> )
                            -
```

where

<file name>, INT:ref,

is an array containing the name of the disc file to be created:

To create a permanent disc file, <file name> must be of the form

```
<file name>[0:3]  is $<volume name><blank fill>
                  or \<system number><volume name><blank fill>
<file name>[4:7]  is <subvol name><blank fill>
<file name>[8:11] is <disc file name><blank fill>
```

To create a temporary disc file, <file name> must be of the form

```
<file name>[0:11] is $<volume name><blank fill>
```

When CREATE completes, a temporary file name is returned in <file name>[4:7]. The temporary file can then be opened by passing <file name> to OPEN.

<primary extent size>, INT:value,

if present, is the size of the primary extent in 2048-byte units (maximum extent size is 134,215,680 bytes). If omitted, a primary extent size of one (2048 bytes) is assigned.

→

```
example:

   CALL CREATE ( filename );          ! primary extent size = 1,
                                      ! file code = 0,
                                      ! secondary extent size = 1.

   IF < THEN ...                      ! CREATE failed.
```

CONSIDERATIONS

- File pointer action:

  end-of-file pointer := 0D;

- Execution of the CREATE procedure does not allocate any disc area; it only provides an entry in the volume's directory indicating that the file exists.

- CREATE does not provide access to the new file; the OPEN procedure must be called.

- If the CREATE fails (i.e., condition code other than CCE returned), the reason for the failure can be determined by calling the file system FILEINFO procedure and passing -1 as the <file number> parameter.

- The file is created with the user's default security. A file's security can be altered by opening the file and issuing the appropriate SETMODE functions.

- An unstructured disc file can be created for either even unstructured or odd unstructured access. On reads to and writes from even unstructured files, odd read counts and write counts are rounded to the next even number (3 becomes 4, 5 becomes 6, etc.); and a POSITION for such a file must be to an even byte address. An odd unstructured file permits reads and writes of odd byte counts and positioning to an odd byte address. If <file type>.<13:15> passed to CREATE is all zeros (specifying an unstructured file) and <file type>.<12> is 0, an even unstructured file is created. If <file type>.<13:15> is all zeros and <file type>.<12> is 1, an odd unstructured file is created. (If the FUP CREATE command is used to create the file, it will create an even unstructured file unless the ODDUNSTR parameter is given.)

Table 2-3. Device Types and Subtypes

device type, <device type>.<4:9>,			device subtype, <device type>.<10:15>,		
0	=	Process	0		
1	=	Operator Console	0		
2	=	$RECEIVE	0		
3	=	Disc	0	=	10 MB capacity (NonStop systems only)
		(Note: For discs, <device type>.<0> = 1 denotes a removable disc volume; <device type>.<1> = 1 denotes a TMF audited disc volume.)	1	=	50 MB capacity (NonStop systems only)
			2	=	160 MB capacity (NonStop systems only)
			3	=	240 MB capacity
			4	=	64 MB capacity (P/N 4105, 4106)
			5	=	64 MB capacity, movable head portion (P/N 4109)
			6	=	540 MB capacity (P/N 4116) (NonStop II systems only)
			7	=	1.45 MB capacity, fixed head portion (P/N 4109) (NonStop systems only)
			8	=	128 MB capacity (P/N 4110, 4111)
4	=	Magnetic Tape	0	=	Nine-Track
			1	=	Seven-Track
			2	=	Tri-Density Tape Drive (P/N 5106)
5	=	Line Printer	0	=	Belt Printer
			1	=	Drum or Band
			2	=	Current-Loop, Belt Type
			3	=	Matrix Serial (P/N 5508)
			4	=	Matrix Serial (P/N 5520)
			5	=	Band, extended char. set
6	=	Terminal (conversational or page mode)	0	=	Conversational Mode
			1	=	Page Mode (6511, 6512)
			2	=	Page Mode (6520, 6524)
			3	=	Page Mode (Remote 6520)
			4	=	Page mode (6530)
			5	=	Page mode (Remote 6530)
			6-10	=	Conversational Mode (various screen sizes)
			32	=	Hard-Copy Console

Table 2-3. Device Types and Subtypes (cont'd)

device type, <device type>.<4:9>,			device subtype, <device type>.<10:15>,		
59	=	AXCESS Data Communication Line	0	=	AM6520 Access Method
60	=	AXCESS Data Communication Line	0 1	= =	AM3270 Access Method TR3271 Access Method
61	=	AXCESS Data Communication Line	0	=	X25AM Access Method (any subtype 0-63 is accepted with no effect)
62	=	EXPAND Network Control Process (NCP)	0		
63	=	EXPAND Line Handler	0 1 2	= = =	Single-Line Path Path Entry, Multi-Line Path Line Entry, Multi-Line Path

```
    <buffer length>, INT:value,

      is the length, in bytes, of the <buffer> array.  This
      specifies the maximum number of characters in the text line
      that will be transferred into <buffer>.

    <sequence number>, INT(32):ref,

      is the sequence number multiplied by 1000, in double-word
      integer form, of the text line just read.

  example:

    count := EDITREAD(control^block, line, length, seq^num);
```

The following extended example illustrates the use of EDITREADINIT and EDITREAD.

The data is declared as follows:

```
    .
  LITERAL buf^size = 512, !EDITREAD's internal buffer size in bytes
          length   = 80;  !length of the application's buffer (bytes)

  INT   fnum,
        fcode,
        error,
        count,
       .control^block[0:(39+buf^size/2)];  ! global data declaration.

  STRING  .line[0:length-1];               ! application's buffer.

  INT(32) seq^num;
```

First the text file is opened and verified that it is an edit format file:

```
    .
    .
  CALL OPEN(fname,fnum,...);
  IF < THEN ...;
  CALL FILEINFO(fnum,,,,,,,,fcode);
  IF fcode <> 101 THEN ... ! not edit format file.
    .
    .
```

Then EDITREADINIT is called to initialize the edit control block and specify EDITREAD's internal buffer size:

For example:

```
INT control^block[0:(39+buf^size/2)],
    position[0:2];
  .
  .
! EDITREADINIT and one or more EDITREADs are called
  .
  .
position ':=' control^block[1] FOR 3;     ! save current position
  .
  .
! more EDITREADs
  .
  .
control^block[1] ':=' position FOR 3;     ! restore saved position
control^block.<0> := 1;

! next EDITREAD returns same record returned after position
!    was saved
```

```
example:

   INT .control^block[0:(39+256/2)];
   n := EDITREADINIT(control^block, fnum, 256);
```

An extended example using both EDITREADINIT and EDITREAD is given in the syntax description of the EDITREAD procedure.

entered (signalling that the condition cannot be corrected), FILEERROR returns a zero indicating that the operation should not be retried. If any other data is entered (typically, carriage return), it signals that the condition has been corrected, and FILEERROR returns a one, indicating that the operation should be retried.

- If the error is caused by an ownership error (error 200) or a path down error (error 201) and the alternate path is operable, FILEERROR returns a one, indicating that the operation should be retried. If the alternate path is inoperable, a zero is returned.

- Any other error results in the file name being printed on the home terminal, followed by the file system error number. A zero is returned, indicating that the operation should not be retried.

An example:

```
error := 1;
WHILE error DO
  BEGIN
    CALL WRITE(fnum,buffer,count);
    IF < THEN
      BEGIN
        IF NOT FILEERROR(fnum) THEN CALL ABEND;
      END
    ELSE error := 0;
  END;
```

It may be desirable to check for certain errors before calling FILEERROR. In this case, the program itself should first call FILEINFO. For example:

The FILEINFO procedure is used to obtain error and characteristic information about an open file.

The call to the FILEINFO procedure is:

```
CALL FILEINFO ( <file number>
---- -------- - -------------
                , <error>
                , <file name>
                , <logical device number>
                , <device type>
                , <extent size>
                , <end-of-file location>
                , <next-record pointer>
                , <last mod time>
                , <file code>
                , <secondary extent size>
                , <current-record pointer>
                , <open flags> )
                                -
```

where

<file number>, INT:value,

identifies the file whose characteristics are to be returned.

<error>, INT:ref:1,

if present, is returned the error number associated with the last operation on the file (see "Errors and Error Recovery").

<file name>, INT:ref:12,

if present, is returned the file name of this file. See "File Names" for the file name format.

<logical device number>, INT:ref:1,

if present, is returned the logical device number of the device where this file resides (in binary). (If your files are partitioned, use the value 16 instead of 1.)

<device type>, INT:ref:1,

if present, is returned the device type of the device associated with this file. See "DEVICEINFO Procedure", table 2-3.

→

<open flags>.<12:15> is the maximum number of concurrent no-wait i/o operations that can be in progress on this file at any given time. <open flags>.<12:15> = 0 implies wait i/o.

<open flags>.<9:11> is the exclusion mode:

```
0 = shared access
1 = exclusive access
3 = protected access
```

<open flags>.<8> = 1 indicates that, for process files, the OPEN message is to be sent no-wait.

On NonStop systems only, <open flags>.<6> = 1 indicates that resident buffers have been provided by the application process for calls to file system i/o routines (see "OPEN Procedure" and "File System Advanced Features").

<open flags>.<3:5> is the access mode:

```
0 = read/write access
1 = read-only access
2 = write-only access
```

<open flags>.<1> = 1, for the $RECEIVE file only, means that the process wants to receive OPEN, CLOSE, CONTROL, SETMODE, RESETSYNC, and CONTROLBUF system messages.

condition code settings:

<	(CCL)	indicates that an error occurred; the error number is returned in <error>.
=	(CCE)	indicates that FILEINFO executed successfully.
>	(CCG)	is not returned by FILEINFO.

example:

```
CALL FILEINFO ( infile, err^num );
```

CONSIDERATIONS

- The error number of a preceding AWAITIO on any file or waited OPEN that failed can be obtained by passing a -1 in the <file number> parameter. The error number is returned in <error>.
- 32 is returned in <error> (if <error> is a parameter present in the call) if a process has never opened any files and -1 is specified in the <file number> parameter.

The FNAMECOLLAPSE procedure converts a file name from its internal to its external form. The system number of a network file name is converted to the corresponding system name.

The call to the FNAMECOLLAPSE procedure is:

```
{ <length> := } FNAMECOLLAPSE ( <internal name>
{ CALL       } ------------- - ---------------
                               , <external name> )
                               - --------------- -
```

where

<length>, INT,

is returned the number of bytes in <external name>.

<internal name>, INT:ref:12,

is the name to be converted. If this is in local form, it is converted to external local form; if it is in network form, it is converted to external network form. Network file names are discussed in the "File Names" section.

<external name>, STRING:ref:26 or STRING:ref:34

contains, on return, the external form of <internal name>. If <internal name> is a local file name, <external name> contains 26 bytes; if a network name is converted, <external name> contains 34 bytes.

example:

```
length := FNAMECOLLAPSE ( internal, external );
```

Examples:

local: $SYSTEM SUBVOL MYFILE
is converted to "$SYSTEM.SUBVOL.MYFILE"

network: \<sysnum>SYSTEMSUBVOL MYFILE
is converted to "\<system name>.$SYSTEM.SUBVOL.MYFILE

The FNAMECOMPARE procedure compares two file names within a local or network environment to determine whether these file names refer to the same file or device. For example, one name may be a logical system name or a device number, while the other reference is a symbolic name. The file names compared must be in the standard twelve-word internal format that is returned by FNAMEEXPAND.

The call to the FNAMECOMPARE procedure is:

```
{ <status> := } FNAMECOMPARE ( <file name 1> , <file name 2> )
{ CALL       } ------------ - ------------- - ------------- -
```

where

<status>, INT,

is a value indicating the outcome of the comparison. Values for <status> are:

-1	=	(CCL): the file names do not refer to the same file.
0	=	(CCE): the file names refer to the same file.
+1	=	(CCG): the file names refer to the same volume name, device name, or process name on the same system; however, words [4:11] are not the same: <file name 1>[4] <> <file name 2>[4] FOR 8.

A value less than -1 is the negative of a file system error code. This indicates that the comparison is not attempted due to this error condition.

That value returned from the program function determines the condition code setting.

<file name 1>, INT:ref:12,

is the first comparable file name. Each <file name> array may contain either a local file name or a network file name. Definitions of file names are found in the "File Names" section.

<file name 2>, INT:ref:12,

is the second comparable file name.

condition code settings:

See <status> parameter.

In a non-network system, execution of the example just given returns a status of -1 and the condition code (CCL).

Whether a system is a network node or not, execution of

```
fname1 ':=' [ "$SERVR  #START  UPDATING" ];
fname2 ':=' [ "$SERVR  #FINISH UPDATING" ];
status := FNAMECOMPARE ( fname1, fname2 );
```

returns a status of +1 and the condition code (CCG).

In any system, execution of

```
fname1 ':=' [ "$0013 ", 9 * [ "  "] ];
fname2 ':=' [ "$DATAX", 9 * [ "  "] ];
status := FNAMECOMPARE ( fname1, fname2 );
```

returns a status of zero and condition code (CCE) if the device name $DATAX is defined as logical device number 13 at SYSGEN time; otherwise, a status of -1 and the condition code (CCL) is returned.

FNAMECOMPARE can also verify the specified file names as follows:

```
! assume all variables and procedures have been
! properly defined and initialized elsewhere
!
! also assume  LITERAL  legal = 0;

IF FNAMEEXPAND ( external^name, internal^name, default^names ) THEN
   BEGIN
   ! something reasonable was entered.
   IF FNAMECOMPARE ( internal^name, internal^name ) = legal THEN
      ! it may not exist, but looks okay.
      BEGIN
        .
        ! normal processing.
        .
      END
   ELSE
      ! the format is not legal.
      BEGIN
        .
        ! error processing.
        .
      END;
   END;
```

```
    <internal file name>, INT:ref,

      is an array of twelve words where FNAMEEXPAND returns the
      expanded file name.  This cannot be the same array as
      <external file name>.

    <default names>, INT:ref,

      is an array of eight words containing the default volume and
      subvol names to be used in file name expansion.  <default
      names> is of the form:

        <default names>[0:3]  = default <volume name> (blank
                                filled on right)
        <default names>[4:7]  = default <subvol name> (blank
                                filled on right)

      <default names>[0:7] corresponds directly to <word>[1:8] of
      the Command Interpreter param message.  See section 11,
      "COMINT/Application Interface", for the param message format.

  example:

    length := FNAMEEXPAND(inname,outname,pmsg[1]);
```

FNAMEEXPAND converts local file names to local names, and network file names to network names. Network file names are described under "File Names", section 2.2. When network file names are involved, in addition to expanding the local part of the name using the defaults, FNAMEEXPAND converts the system name to the appropriate system number. (If the system name is unknown, FNAMEEXPAND supplies 255 for the system number.)

FNAMEEXPAND expands file names as follows:

<disc file name> is returned as

```
   <file name>[0:3]  = $<default volume name><blank fill>
   <file name>[4:7]  = <default subvol name><blank fill>
   <file name>[8:11] = <disc file name><blank fill>
```

<subvol name>.<disc file name> is returned as

```
   <file name>[0:3]  = $<default volume name><blank fill>
   <file name>[4:7]  = <subvol name><blank fill>
   <file name>[8:11] = <disc file name><blank fill>
```

```
.
SCAN ext^name WHILE " " -> @p; ! skip leading blanks.
@p := FNAMEEXPAND(p, infile, defaults) + @p;
```

on the completion of FNAMEEXPAND, <infile> contains

```
"$voll   svoll   filea   "
```

which is suitable for passing to the file system CREATE, OPEN, RENAME, and PURGE procedures, as well as the process control procedures NEWPROCESS and NEWPROCESSNOWAIT.

"p" is incremented by the number of characters in the external file name.

```
.
SCAN p WHILE " " -> @p; ! skip intermediate blanks.
CALL FNAMEEXPAND(p, outfile, defaults);
.
```

on the completion, "outfile" contains

```
"$system svoll   fileb   ".
```

Another example:

Suppose that system \NEWYORK is assigned system number 4. Then the external file name "\NEWYORK.$DATA.SUB.MYFILE" is converted by FNAMEEXPAND to

```
\<%4>DATA  SUB     MYFILE
```

where "<%4>" denotes 4 in the second byte.

The use of FNAMEEXPAND in programming network applications is discussed further in the EXPAND Users Manual.

<device name>, INT:ref:4,

is returned the device name or volume name of the designated device if it exists, or the next higher logical device if the designated device does not exist. If end of LDT is encountered, <device name> is unchanged.

<system number>, INT,

if present, specifies the system (in a network) whose Logical Device Table is to be searched for <logical device no.>.

If absent, the local system is assumed.

condition code settings:

The condition code setting has no meaning following a call to GETDEVNAME.

example:

```
! get the names of all logical devices.
ldev := 0;
WHILE NOT GETDEVNAME ( ldev , devname ) DO
  BEGIN
    CALL print ( ldev , devname );
    ldev := ldev + 1;
  END;
```

CONSIDERATIONS

- If the device specified by <logical device no> is remote, its device name is returned in network form; otherwise, the device name is returned in local form.

  If the <system number> parameter is supplied, devices whose names contain seven characters are not accessible using this procedure.

The LASTRECEIVE procedure is used to obtain the process ID and/or the message tag associated with the last message read from the $RECEIVE file. This information is contained in the file's main-memory resident Access Control Block; therefore, the application process is not suspended because of a call to LASTRECEIVE.

Note: A call to LASTRECEIVE must immediately follow the call to READUPDATE of $RECEIVE or the AWAITIO that completes it. Otherwise, the information returned may be invalid.

The call to the LASTRECEIVE procedure is:

```
CALL LASTRECEIVE ( <process id> , <message tag> )
---- ----------- -                              -
```

where

<process id>, INT:ref:4,

if present, is returned the ID of the process that sent the last message read through the $RECEIVE file. If the process is in the PPD, the information returned consists of

```
<process id>[0:2] = $<process name>
<process id>[3]   = <cpu,pin>
```

If the process is not in the PPD, the information returned consists of

```
<process id>[0:2] = <creation time stamp>
<process id>[3]   = <cpu,pin>
```

<message tag>, INT:ref:1,

is used when the application process performs message queueing. If present, <message tag> is returned a value that identifies the request message just read among other requests currently queued. To associate a reply with a given request, <message tag> is passed in a parameter to the REPLY procedure. The value of <message tag> will be the lowest integer between zero and <receive depth> - 1, inclusive, that is not currently being used as a message tag. When a reply is made, its associated message tag value is made available for use as a message tag for a subsequent request message.

→

The LOCATESYSTEM procedure provides the system number corresponding to a system name, and returns the logical device number of the line handler controlling the path to a given system.

The call to the LOCATESYSTEM procedure is:

```
{ <ldev> := }  LOCATESYSTEM ( <system number> , <system name> )
{ CALL      }  ------------ - ---------------                  -
```

where

<ldev>, INT,

returns a value as follows:

-1 = all paths to the specified system are down.
 0 = the system is not defined.
>0 = the logical device number of the line handler in the specified system.

<system number>, INT:ref,

if <system name> is provided, is returned the system number corresponding to <system name>. If <system name> is not provided, <system number> should contain the system number to be located.

<system name>, INT:ref:4,

if present, specifies the system to be located, and causes the corresponding system number to be returned in <system number>.

example:

```
ldev := LOCATESYSTEM( sys^num, sys^name );
IF NOT ldev THEN ... ! trouble
```

CONSIDERATIONS

- Note that if <system name> is provided by the caller, <system number> is returned the corresponding number; but if <system name> is omitted, <system number> must be provided by the caller.

condition code settings:

```
<  (CCL) indicates that an error occurred (call FILEINFO).
=  (CCE) indicates that the LOCKFILE was successful.
>  (CCG) indicates that the file is not a disc file.
```

example:

```
CALL LOCKFILE ( file^num );
IF < THEN .....;          ! error
```

CONSIDERATIONS

- Locks are granted on an open file (i.e., file number) basis. Therefore, if a process has multiple opens of the same file, a lock of one file number excludes accesses to the file through other file numbers.

- If a call to CONTROL, WRITE, or WRITEUPDATE is made and the file is locked but not through the file number supplied in the call, the call is rejected with a "file is locked" error indication (error 73).

- If the default locking mode is in effect when a call to READ or READUPDATE is made and the file is locked but not locked through the file number supplied in the call, the caller of READ or READUPDATE is suspended and queued in the "locking" queue behind other processes attempting to lock or read the file.

  Note that a deadlock condition occurs if a call to READ or READUPDATE is made by the process having a file locked but not via the file number supplied to READ or READUPDATE.

- If the alternate locking mode is in effect when READ or READUPDATE is called and the file is locked but not through the file number supplied in the call, the call is rejected with a "file is locked" error indication (error 73).

- The locking mode is specified via the SETMODE procedure, function 4.

- Locks are not nested.

  For example:

```
    .
CALL LOCKFILE ( file^a );
    .
  "file^a" becomes locked.
```

The MONITORNET procedure enables/disables receipt of system messages concerning the status of processors in remote systems.

The call to the MONITORNET procedure is:

```
CALL MONITORNET ( <enable> )
---- ---------- - -------- -

where

   <enable>, INT,

      has the following meaning:

         0 =  disable receipt of messages.
         1 =  enable receipt of messages.

 example:

    CALL MONITORNET ( 1 );
```

CONSIDERATIONS

- A process that has enabled MONITORNET receives a system message via $RECEIVE whenever a change in the status of a remote processor occurs. The format of this message is:

  ```
  word[0]:          -8
  word[1].<0:7>:    system number
  word[1].<8:15>:   number of cpu's in system
  word[2]:          current processor status bit mask
  word[3]:          previous processor status bit mask
  ```

  The processor status bit masks have a one in bit <cpu number> to indicate that the processor is up, and a zero to indicate that the processor is down.

- MONITORNET only provides notification of status changes for remote processors. To receive notification of status changes for local processors, an application process must still call MONITORCPUS.

The NEXTFILENAME procedure is used to obtain the names of disc files on a designated volume. NEXTFILENAME returns the next file name in alphabetical sequence after the file name supplied as a parameter. The intended use of NEXTFILENAME is in an iterative loop, where the file name returned in one call to NEXTFILENAME is used to specify the starting point for the alphabetical search in the subsequent call to NEXTFILENAME. In this manner, a volume's file names are returned to the application process in alphabetical order through succeeding calls to NEXTFILENAME.

The call to the NEXTFILENAME procedure is:

```
<error> := NEXTFILENAME ( <file name> )
           ------------ - ----------- -

where

   <error>, INT

     is a file system error number indicating the outcome of
     the call.  Common error number returns are

        0 = no error, next file name in alphabetical sequence is
            returned in <file name>.
        1 = end-of-file, there is no file in alphabetical
            sequence following the file name supplied in <file
            name>.
       13 = illegal file name specification.

   <file name>, INT:ref:12,

     on the call, is passed the file name from which search for
     the next file name begins.  <file name> on the initial call
     can be one of the following forms:

       <file name>[0:11] = $<volume name><blank fill>
                    or \<system number><volume name><blank fill>

         This form is used to obtain the name of the first file
         on $<volume name>.

       <file name>[0:3]  = $<volume name><blank fill>
                    or \<system number><volume name><blank fill>

       <file name>[4:11] = <subvol name><blank fill>

         This form is used to obtain the name of the first file
         in <subvol name> on $<volume name>.
```

→

The OPEN procedure establishes a communication path between an application process and a file. When OPEN completes, a "file number" is returned to the application process. The file number identifies this access to the file in subsequent file system calls.

The call to the OPEN procedure is:

```
CALL OPEN ( <file name> , <file number>
---- ---- - ----------- - -------------
          , <flags>
          , <sync or receive depth>
          , <primary file number> , <primary process id> )
                                                          -
```

where

<file name>, INT:ref,

is an array containing the name of the file to be opened (see "File Names").

<file number>, INT:ref:1,

is returned from OPEN and is used to identify the file in subsequent file system calls.

<flags>, INT:value,

if present, specifies certain attributes of the file. If omitted, all fields are set to zero. The bit fields in the <flags> parameter (<flags>.<0> being the leftmost, or high-order, bit) are defined as follows:

<flags>.<0> = unused; must be 0

<flags>.<1> = opener wishes to receive OPEN, CLOSE, CONTROL, SETMODE, RESETSYNC, and CONTROLBUF messages ($RECEIVE only)
0 = no 1 = yes
(must be 0 for all other files)

<flags>.<2> = unstructured access
(ENSCRIBE structured files only)
0 = no 1 = yes
(must be 0 for all other files)

<flags>.<3> = unused; must be 0

<flags>.<4> } = access mode
<flags>.<5> }
0 = read/write 1 = read-only
2 = write-only

→

For write to a process pair,

the state of this parameter indicates whether or not a write to a process pair is automatically redirected to the backup process if the primary process or its processor module fails. If this parameter >= 1, then a write is automatically redirected in a manner invisible to the originator of a message. If this parameter = 0 and a write cannot occur to the primary process of a process pair, an error indication is returned to the message's originator. On a subsequent retry by the originator, the file system will redirect the write to the backup process.

For other device types, this parameter is ignored.

The next two parameters are supplied only if the open is by the backup process of a process pair, the file is currently open by the primary process, and the Checkpointing Facility (described in section five) is not used.

<primary file number>, INT:value,

is the file number returned to the primary process when it opened this file.

<primary process id>, INT:ref,

is an array which contains the <process id> of the corresponding primary process. The primary process must already have the file open.

condition code settings:

<	(CCL) indicates that the OPEN failed (call FILEINFO).
=	(CCE) indicates that the file opened successfully.
>	(CCG) is not returned from OPEN.

example:

```
CALL OPEN ( filename, filenum ); ! wait i/o, exclusion mode
                                 ! = shared, access mode =
                                 ! read/write, sync depth = 0.
IF < THEN ....                   ! OPEN failed.
```

For a non-process or waited (no-wait depth = 0) file, <flags>.<8> is reset to zero internally and ignored. A call to FILEINFO after the call to OPEN can return the value of the internal flags; if bit 8 = 1, then a call to AWAITIO must be performed to complete the open.

When a process file is opened in a no-wait manner, that file is checkopened in a no-wait manner. See "CHECKOPEN Procedure", section 5 in this manual, for further discussion of no-wait checkopens. Otherwise, all errors are returned by OPEN.

- See "Errors and Error Recovery" for considerations when using no-wait i/o.

- When a disc file open is attempted, a file security check takes place. The accessor's (i.e., caller's) security level is checked against the file's security level for the requested access mode. (File security is set via the SETMODE procedure or the File Utility Program, FUP, SECURE Command.) If the caller's security level is equal to or higher than the file's security level for the requested access mode, then the caller passes the security check. If the caller fails the security check, the open fails, and a subsequent call to FILEINFO returns error 48. See figure 2-15.

Table 2-4. Exclusion/Access Mode Checking

OPEN ATTEMPTED WITH			FILE CURRENTLY OPEN WITH								
Exclusion Mode		CLOSED	S	S	S	E	E	E	P	P	P
	Access Mode		R/W	R	W	R/W	R	W	R/W	R	W
S	R/W	Y	Y	Y	Y						
S	R	Y	Y	Y	Y				Y	Y	Y
S	W	Y	Y	Y	Y						
E	R/W	Y	ALWAYS FAILS								
E	R	Y									
E	W	Y									
P	R/W	Y		Y							
P	R	Y		Y						Y	
P	W	Y		Y							

Exclusion Mode:
- S = Sharable
- E = Exclusive
- P = Protected

Access Mode:
- R/W = Read/Write
- R = Read only
- W = Write only

Y = Yes, OPEN successful
Blank = No, OPEN fails.

Notes:
- BACKUP opens the file currently being backed up with R, P.
- BACKUP with "OPEN" option specified opens the file with R, S.
- RESTORE opens the file currently being restored with R/W, E.
- When a program file is running it is opened with the equivalent to R, P.

For unstructured disc files, The POSITION procedure is used to set the current-record and next-record pointers to a specific address in the file. Any subsequent i/o transfer will begin from that point. Note that the caller is not suspended because of a call to POSITION.

A call to the POSITION procedure will be rejected with an error indication if there are any outstanding no-wait operations pending on the specified file.

The call to the POSITION procedure is:

```
CALL POSITION ( <file number> , <record specifier> )
---- -------- - ------------ - ------------------ -

where

   <file number>, INT:value,

     identifies the file to be positioned.

   <record specifier>, INT(32):value,

     is a relative byte address that specifies the new
     setting for the current-record and next-record pointers.
     For even unstructured files, this must be an even byte
     address (see CREATE procedure), or the operation fails
     with error 2 (operation invalid for this type file).

     Specifying -1D indicates that subsequent writes should occur
     at the end-of-file location (i.e., until a new record
     specifier is supplied).

condition code settings:

   <  (CCL) indicates that an error occurred (call FILEINFO).
   =  (CCE) indicates that the POSITION was successful.
   >  (CCG) indicates no operation; not a disc file.

example:

   CALL POSITION ( infile, 1000D );
   IF < THEN ....                    ! error occurred
```

CONSIDERATIONS

- File pointer action:

  current-record pointer := next-record pointer :=
  if rba = -1D then end-of-file pointer else rba

The PURGE procedure is used to delete a closed disc file. When PURGE is executed, the disc file name is deleted from the volume's directory, and any space previously allocated to that file is made available to other files.

The call to PURGE is:

```
CALL PURGE ( <file name> )
---- ----- - ----------- -

where

   <file name>, INT:ref,

   is an array containing the name of the disc file to be purged:

   To purge a permanent disc file, <file name> must be of the form

         <file name>[0:3]  is $<volume name><blank fill>
                     or \<system number><volume name><blank fill>
         <file name>[4:7] is <subvol name><blank fill>
         <file name>[8:11] is <disc file name><blank fill>

   To purge a temporary disc file, <file name> must be of the form

         <file name>[0:3]  is $<volume name><blank fill>
                     or \<system number><volume name><blank fill>
         <file name>[4:11] is <temporary file name>

condition code settings:

   <  (CCL) indicates that the PURGE failed (call FILEINFO).  Note,
            however, that in the case of a disc free space error
            (such as File System Errors 52, 54, 58), the file will
            be purged and an error will be returned.
   =  (CCE) indicates that the file was purged successfully.
   >  (CCG) indicates that the device is not a disc.

example:

   CALL PURGE ( oldfilename );
   IF < THEN ...                        ! PURGE failed.
```

CONSIDERATIONS

- If PURGE fails, the reason for the failure can be determined by calling FILEINFO, passing -1 as the <file number> parameter.
- If the file is a TMF audited file and there are pending transaction mode record or file locks, the purge fails with file error 12, whether or not openers of the file still exist.

```
condition code settings:

   <  (CCL) indicates that an error occurred (call FILEINFO).
   =  (CCE) indicates that the READ was successful.
   >  (CCG) for disc and non-disc devices, indicates that the
            end-of-file was encountered; for the $RECEIVE file, a
            system message was received. (Call FILEINFO.)

example:

   CALL READ ( filenum, inbuffer, 72 );
   IF < THEN ....                    ! READ failed.
```

CONSIDERATIONS

- If a "wait" read is executed, the <count read> parameter indicates the number of bytes actually read.
- If a "no-wait" read is executed, <count read> has no meaning and can be omitted. The count of the number of bytes read is obtained when the i/o operation completes via the <count transferred> parameter of the AWAITIO procedure.
- If the read is from a non-disc device, the right half of the last word of an odd count transfer will be garbage.

CONSIDERATIONS FOR DISC FILES

- For a read from an unstructured disc file, data transfer begins at the position indicated by the next-record pointer.
- <count read> determination

```
<count read> := $MIN ( <read count> , end-of-file pointer
                                     - next-record pointer )
```

- File pointer action:

```
CCG := if next-record pointer = end-of-file pointer then 1
                                                  else 0;
current-record pointer := next-record pointer;
next-record pointer := next-record pointer + <count read>;
```

- If the read is from an even unstructured disc file, the value of <read count> is rounded up to an even number (see CREATE procedure).

The READUPDATE procedure is used to read data from a disc or interprocess file in anticipation of a subsequent write to the file.

For disc files, READUPDATE is used for random processing. Data is read from the file at the position of the current-record pointer. A call to this procedure typically follows a corresponding call to POSITION. The values of the current- and next-record pointers are not changed by the call to READUPDATE.

For interprocess communication, READUPDATE is used to read a message from the $RECEIVE file that will be replied to in a later call to REPLY. Each message read via READUPDATE must be replied to in a corresponding call to REPLY.

If the READUPDATE procedure is being used to initiate an operation with a file opened with no-wait i/o specified, the operation must be completed with a corresponding call to the AWAITIO procedure.

The call to the READUPDATE procedure is:

```
CALL READUPDATE ( <file number> , <buffer> , <read count>
                , <count read>
                , <tag> )
                        -
```

where

<file number>, INT:value,

identifies the file to be read.

<buffer>, INT:ref:*,

is an array where the information read from the file is returned.

<read count>, INT:value,

is the number of bytes to be read: {0:4096} for disc files, {0:32000} for $RECEIVE.

<count read>, INT:ref:1,

for wait i/o only, if present, is returned a count of the number of bytes returned from the file into <buffer>.

→

via the file number supplied to READUPDATE.

- If the alternate locking mode is in effect when READUPDATE is called and the file is locked but not through the file number supplied in the call, the call is rejected with a "file is locked" error indication (error 73).

- The locking mode is specified via the SETMODE procedure, function 4.

CONSIDERATIONS FOR INTERPROCESS COMMUNICATION

- Each message read in a call to READUPDATE, including system messages, must be replied to an a corresponding call to the REPLY procedure.

- Several interprocess messages can be read and queued by the application process before a reply need be made. The maximum number of messages that the application process expects to read before a corresponding reply is made must be specified in the <receive depth> parameter to the OPEN procedure.

- If more than one message is to be queued by the application process (i.e., <receive depth> > 1), a message tag that is associated with each incoming message must be obtained in a call to the LASTRECEIVE procedure following each call to READUPDATE. To direct a reply back to the originator of the message, the message tag that was associated with the incoming message is passed back to the system in a parameter to the REPLY procedure. If messages are not to be queued, it is not necessary to call LASTRECEIVE.

```
      inclusive, that is not currently being used as a message
      tag.  When a reply is made, its associated message tag value
      is made available for use as a message tag for a subsequent
      request message.

   <sync id>, INT(32):variable,

      if present, is returned the sync ID associated with this
      message.

   <file number>, INT:ref:1,

      if present, is returned the file number of the file in the
      requesting process that is associated with this message.

   <read count>, INT:ref:1,

      if present, is returned the number of bytes requested in
      reply to the message.  If the message is the result of a
      request made in a call to WRITE, <read count> will be zero
      (0).  If the message is the result of a request made in a
      call to WRITEREAD, <read count> will be the same as the
      read count value passed by the requestor to WRITEREAD.

condition code settings:

   <  (CCL) indicates that $RECEIVE is not open.
   =  (CCE) indicates that RECEIVEINFO was successful.
   >  (CCG) is not returned by RECEIVEINFO.

example:

   CALL RECEIVEINFO ( req^syncid , req^fnum , req^readcount );
   IF < THEN ....
```

CONSIDERATIONS

- The process ID returned by RECEIVEINFO following receipt of a preceding OPEN, CLOSE, CONTROL, SETMODE, RESETSYNC, or CONTROLBUF system message identifies the process associated with the operation.
- The high-order three words of the process ID are zero following the receipt of system messages other than OPEN, CLOSE, CONTROL, SETMODE, RESETSYNC, and CONTROLBUF.
- A sync ID is a doubleword, unsigned integer. Each process file that is open has its own sync ID. Sync ID's are not part of the message data; rather, the sync ID value associated with a

The REFRESH procedure is used to write control information contained in File Control Blocks (FCB's), such as the end-of-file pointer, to the associated physical disc volume. (While a file is open, its control information is kept in its main-memory resident FCB; this control information is normally written to the physical volume only when the last process having the file open closes the file.) This procedure or the equivalent Peripheral Utility Program (PUP) REFRESH command should be performed for all volumes prior to a total system shutdown.

For further information, see the section on unstructured disc files in the ENSCRIBE Programming Manual.

The call to the REFRESH procedure is:

```
{ <error> := } REFRESH ( <volume name> )
{  CALL     } -------
```

where

<error>, INT,

is a file system error number indicating the outcome of the call.

<volume name>, INT:ref:4,

is either

$<volume name> or \<system number><volume name>

<volume name> specifies a volume whose associated FCB's should be written to disc. $<volume name> can be specified as a full twelve-word <file name>; <file name>[4:11] is ignored.

If omitted, all FCB's for all volumes are written to their respective discs.

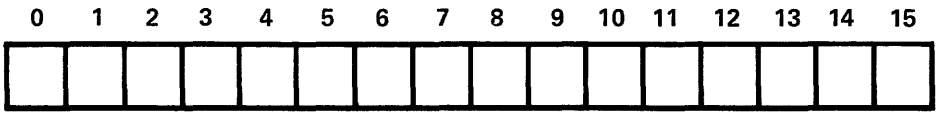
example:

```
CALL REFRESH;
```

CONSIDERATIONS

- When REFRESH is called without a <volume name>, the error return is always zero.

- The bits in the low-order word are ordered from 0 to 15, from left to right:



- REMOTEPROCESSORSTATUS can also be used to obtain the status of local processors:

```
INT(32) my^processor^status;
my^processor^status := REMOTEPROCESSORSTATUS( MYSYSTEMNUMBER );
```

The REPLY procedure is used to send a reply message, via the $RECEIVE file, to a message that was received earlier in a corresponding call to READUPDATE.

The REPLY procedure can be called even if there are any outstanding no-wait i/o operations pending on $RECEIVE.

The call to the REPLY procedure is:

```
CALL REPLY ( <buffer> , <write count>
---- -----   , <count written>
             , <message tag>
             , <error return>  )
```

where

<buffer>, INT:ref,

if present, is an array containing the reply message.

<write count>, INT:value,

if present, is the number of bytes to be written: {0:32000}. If omitted, no data is transferred.

<count written>, INT:ref:1,

if present, is returned a count of the number of bytes written to the file.

<message tag>, INT:value,

is the <message tag> returned from LASTRECEIVE that associates this reply with a message that was previously received. This parameter can be omitted if message queueing is not performed by the application process (i.e., OPEN procedure <receive depth> = 1).

<error return>, INT:value,

if present, is an error indication that is returned to the originator associated with this reply when the originator's i/o operation completes. This indication appears to the originator as though it is a normal file system error return. The originator's condition code is set according to the relative value of <error return>:

→

The REPOSITION procedure is used to position a disc file to a saved position (the positioning information having been saved by calling the SAVEPOSITION procedure). The REPOSITION procedure passes the positioning block obtained via SAVEPOSITION back to the file system. Following a call to REPOSITION, the disc file is positioned to the point where it was when SAVEPOSITION was called.

A call to the REPOSITION procedure will be rejected with an error indication if there are any outstanding no-wait operations pending on the specified file.

The call to the REPOSITION procedure is:

```
CALL REPOSITION ( <file number> , <positioning block> )
---- ---------- - ------------- - ------------------- -

where

   <file number>, INT:value,

     identifies the file to be positioned to a saved position.

   <positioning block>, INT:ref,

     indicates a saved position to be repositioned to.

condition code settings:

   < (CCL) indicates that an error occurred (call FILEINFO).
   = (CCE) indicates that REPOSITION was successful.
   > (CCG) indicates that the file is not a disc file.

example:

   CALL REPOSITION ( file^num, position^block );
   IF < THEN .....;           ! error
```

The SETMODE procedure is used to set device-dependent functions.

A call to the SETMODE procedure will be rejected with an error indication if there are any outstanding no-wait operations pending on the specified file.

The call to the SETMODE procedure is:

```
CALL SETMODE ( <file number> , <function>
               , <parameter 1>
               , <parameter 2>
               , <last params> )
```

where

<file number>, INT:value,

identifies the file to receive the SETMODE <function>.

<function>, INT:value,

is one of the device-dependent functions listed in the "SETMODE Functions" table (table 2-5).

<parameter 1>, INT:value,

is one of the parameters listed in the "SETMODE Functions" table. If omitted, the present value is retained.

<parameter 2>, INT:value,

is one of the parameters listed in the "SETMODE Functions" table. If omitted, the present value is retained.

<last params>, INT:ref:2,

if present, is returned the previous settings of <parameter 1> and <parameter 2> associated with the current function. The format is:

<last params>[0] = old <parameter 1>
<last params>[1] = old <parameter 2> (if applicable)

→

The SETMODENOWAIT procedure is used to set device-dependent functions in a no-wait manner, on no-wait files.

Whereas the SETMODE procedure is a waited operation, and suspends the caller while waiting for a request to complete, the SETMODENOWAIT procedure returns to the caller after initiating a request. A call to SETMODENOWAIT is completed in a call to AWAITIO.

The call to the SETMODENOWAIT procedure is:

```
CALL SETMODENOWAIT ( <file number> , <function>
---- ------------ - ------------ - ----------
                   , <parameter 1>
                   , <parameter 2>
                   , <last params>
                   , <tag> )
                           -
```

where

<file number>, INT:value,

identifies the file to receive the SETMODENOWAIT <function>.

<function>, INT:value,

is one of the device-dependent functions listed in the "SETMODE Functions" table (table 2-5).

<parameter 1>, INT:value,

is one of the <parameter 1> values listed in the "SETMODE Functions" table. If omitted, the present value is retained.

<parameter 2>, INT:value,

is one of the <parameter 2> values listed in the "SETMODE Functions" table. If omitted, the present value is retained.

<last params>, INT:ref:2,

if present, is returned the previous settings of <parameter 1> and <parameter 2> associated with the current <function>.

→

Table 2-5. SETMODE Functions

Note: This table gives only a partial list of SETMODE functions, namely, those used with the i/o devices discussed in this manual. SETMODE functions used with other Tandem software products, such as ENVOY and AXCESS, are described in the manuals for those products.

<function>

1 = disc, set file security:

<parameter 1>

.<0> = 1, for program files only. Set accessor's ID to program file's ID when program file is run.

.<1>, clearonpurge file attribute; if set, clear data in the file before purging the file.

.<4:6>, ID allowed for read.

.<7:9>, ID allowed for write.

.<10:12>, ID allowed for execute.

.<13:15>, ID allowed for purge.

For each of the fields from .<4:6> through .<13.15>, the value may be any one of the following (see section 7 for further information):

0 = any user (local)
1 = member of owner's group (local)
2 = owner (local)
4 = any user (local or remote)
5 = member of owner's community -- i.e., member of owner's group (local or remote)
6 = member of owner's user class -- i.e., owner (local or remote)
7 = super ID only (local)

<parameter 2> is not used.

2 = disc, set file owner ID:

<parameter 1>.<0:7> = group ID
.<8:15> = user ID

<parameter 2> is not used.

Table 2-5. SETMODE Functions (cont'd)

```
 8 = terminal, set system transfer mode (default mode is
     configured):

     <parameter 1> = 0, conversational mode
                   = 1, page mode

     <parameter 2> sets the number of retries of i/o operations.

     Note:  <parameter 2> is used with 6520 terminals only.

 9 = terminal, set interrupt characters:

     <parameter 1>.<0:7>  = character 1
                  .<8:15> = character 2
     <parameter 2>.<0:7>  = character 3
                  .<8:15> = character 4

        (Default for conversational mode is: backspace, line
        cancel, end-of-file, and line termination.  Default for
        page mode is page termination.)

10 = terminal, set parity checking by system (default is
     configured):

     <parameter 1> = 0, no checking
                   = 1, checking

     <parameter 2> is not used.

11 = terminal, set break ownership:

     <parameter 1> = 0, means break disabled (default setting)
                   = <cpu,pin>, means enable break

     and terminal access mode after break is typed:

     <parameter 2> = 0, normal mode (any type file access is
                        permitted)
                   = 1, break mode (only break-type file access
                        is permitted)
```

Table 2-5. SETMODE Functions (cont'd)

22 = line printer (subtype 3 or 4) or terminal, set baud rate:

<parameter 1> = 0, baud rate = 50
1, baud rate = 75
2, baud rate = 110
3, baud rate = 134.5
4, baud rate = 150
5, baud rate = 300
6, baud rate = 600
7, baud rate = 1200
8, baud rate = 1800
9, baud rate = 2000
10, baud rate = 2400
11, baud rate = 3600
12, baud rate = 4800
13, baud rate = 7200
14, baud rate = 9600
15, baud rate = 19200

<parameter 2> is not used.

23 = terminal, set character size:

<parameter 1> = 0, character size = 5 bits
1, character size = 6 bits
2, character size = 7 bits
3, character size = 8 bits

<parameter 2> is not used.

24 = terminal, set parity generation by system:

<parameter 1> = 0, parity = odd
1, parity = even
2, parity = none

<parameter 2> is not used.

25 = line printer (subtype 3), set form length:

<parameter 1> = length of form in lines

<parameter 2> is not used.

Table 2-5. SETMODE Functions (cont'd)

36 = allow requests to be queued on $RECEIVE based on process priority:

<parameter 1>.<15> = 0, use first-in-first-out (FIFO) ordering (default)
1, use process priority ordering

<parameter 2> is not used.

37 = line printer (subtype 1, 4, or 5), get device status:

<parameter 1> is not used.

<parameter 2> is not used.

<last params> = status of device. Status values are:

<last params> for printer (subtype 1 or 5)
(only <last params>[0] is used)

.<5> = DOV, Data overrun	0 = no overrun 1 = overrun occurred
.<7> = CLO, Connector loop open	0 = not open 1 = open (device unplugged)
.<8> = CID, Cable ident	0 = old cable 1 = new cable
.<10> = PMO, Paper motion ** RESERVED FOR LATER USE **	0 = not moving 1 = paper moving
.<11> = BOF, Bottom of form	0 = not at BOF 1 = at bottom
.<12> = TOF, Top of form	0 = not at top 1 = at top
.<13> = DPE, Device parity error	0 = parity OK 1 = parity error
.<14> = NOL, Not on line	0 = on line 1 = not on line
.<15> = NRY, Not ready	0 = ready 1 = not ready

All other bits are undefined.

Table 2-5. SETMODE Functions (cont'd)

37 = line printer (subtype 1, 4, or 5), get device status (cont'd):

<last params>[1] for printer (subtype 4) (cont'd)

.<14:15> } always 3

All other bits are undefined.

38 = Terminal, set special line termination mode and character:

<parameter 1> = 0, set special line termination mode. <parameter 2> is the new line termination character. The line termination character is not counted in the length of a read. No carriage return or line feed is issued (the cursor is not moved) at the end of a read.

= 1, set special line termination mode. <parameter 2> is the the new line termination interrupt character. The line termination character is counted in the length of a read. No carriage return or line feed is issued (the cursor is not moved) at the end of a read.

= 2, reset special line termination mode. The line termination interrupt character is restored to its configured value. <parameter 2> must be present, but is not used.

<parameter 2> = the new line termination interrupt character if <parameter 1> = 0 or 1.

<last params>, if present, returns the current mode in <last params>[0] and the current line termination interrupt character in <last params>[1].

The UNLOCKFILE procedure is used to unlock a disc file that is currently locked by the caller. Unlocking a file allows other processes to access the file. If any processes are queued in the locking queue for the file, the process at the head of the locking queue is granted access and is removed from the queue (the next read or lock request moves to the head of the queue). If the process granted access is waiting to lock the file, it is granted the lock (which excludes other processes from accessing the file) and resumes processing. If the process granted access is waiting to read the file, its read is processed by the file system.

If the UNLOCKFILE procedure is being used to initiate an operation with a file opened with no-wait i/o specified, the operation must be completed with a corresponding call to the AWAITIO procedure.

The call to the UNLOCKFILE procedure is:

```
CALL UNLOCKFILE ( <file number> , <tag> )
--------------- - -------------          -

where

   <file number>, INT:value,

     identifies the file to be unlocked.

   <tag>, INT(32):value,

     for no-wait i/o only, if present, is stored by the system,
     then passed back to the application process by the AWAITIO
     procedure when the unlock operation completes.

condition code settings:

   <  (CCL) indicates that an error occurred (call FILEINFO).
   =  (CCE) indicates that the UNLOCKFILE was successful.
   >  (CCG) indicates that the file is not a disc file.

example:

   CALL UNLOCKFILE ( filenum );
   IF < THEN .....                    ! error occurred.
```

```
example:

   CALL WRITE ( outfile, outbuffer, 72 );
   IF < THEN ....                      ! error occurred.
```

CONSIDERATIONS

- If a "wait" write is executed, the <count written> parameter indicates the number of bytes actually written.
- If a "no-wait" write is executed, <count written> has no meaning and can be omitted. The count of the number of bytes written is obtained when the i/o operation completes via the <count transferred> parameter of the AWAITIO procedure.

CONSIDERATIONS FOR DISC FILES

- If the write is to an unstructured disc file, data is transferred to the record location specified by the next-record pointer. The next-record pointer is updated to point to the record following the record written.
- File pointer action:

```
current-record pointer := next-record pointer;
next-record pointer := next-record pointer + <count written>;
end-of-file pointer := max ( end-of-file pointer,
                             next-record pointer);
```

- If the write is to an even unstructured disc file, the value of <write count> is rounded up to an even number (see CREATE procedure).
- If a call to WRITE is made and the file is locked but not locked through the file number supplied in the call, the call is rejected with a "file is locked" error indication (error 73).
- No block may span more than two extents.

CONSIDERATIONS FOR INTERPROCESS COMMUNICATION

- If the write is to another process, successful completion of the WRITE (or AWAITIO if no-wait) indicates that the destination process is running.

```
    <count read>, INT:ref:1,

      for wait i/o only, if present, is returned a count of the
      number of bytes returned from the file into <buffer>.

    <tag>, INT(32):value,

      for no-wait i/o only, if present, is stored by the system,
      then passed back to the application process by the AWAITIO
      procedure when the write/read operation completes.

condition code settings:

    <  (CCL) indicates that an error occurred (Call FILEINFO).
    =  (CCE) indicates the the WRITEREAD was successful.
    >  (CCG) indicates that control-Y was struck on the terminal.

example:

    CALL WRITEREAD ( termfnum, inout^buffer, 1, 72, num^read );
    IF < THEN ....                    ! error occurred.
```

CONSIDERATIONS

- If a "wait" read is executed, the <count read> parameter indicates the number of bytes actually read.
- If a "no-wait" read is executed, <count read> has no meaning and can be omitted. The count of the number of bytes read is obtained when the i/o operation completes via the <count transferred> parameter of the AWAITIO procedure.

CONSIDERATIONS FOR TERMINALS

- There is no carriage return/line feed sequence sent to the terminal after the write part of the operation.

```
   <tag>, INT(32):value,

     for no-wait i/o only, if present, is stored by the system,
     then passed back to the application process by the AWAITIO
     procedure when the write operation completes.

condition code settings:

   <   (CCL) indicates that an error occurred (call FILEINFO).
   =   (CCE) indicates the the WRITEUPDATE was successful.
   >   (CCG) is not returned by WRITEUPDATE.

example:

   CALL WRITEUPDATE ( outfile, outbuffer, 512 );
   IF = THEN ....                        ! successful.
```

CONSIDERATIONS

- If a "wait" write is executed, the <count written> parameter indicates the number of bytes actually written.
- If a "no-wait" write is executed, <count written> has no meaning and can be omitted. The count of the number of bytes written is obtained when the i/o completes via the <count transferred> parameter of the AWAITIO procedure.

CONSIDERATIONS FOR DISC FILES

- If the write is to an unstructured disc file, data is transferred to the record location specified by the current-record pointer.
- A call to WRITEUPDATE following a call to READ, without intermediate positioning, updates the record just read.
- File pointer action for unstructured files: unaffected.
- If the write is to an even unstructured disc file, the value of <write count> is rounded up to an even number (see CREATE procedure).
- If a call to WRITEUPDATE is made and the file is locked but not locked through the file number supplied in the call, the call is rejected with a "file is locked" error indication (error 73).

The GUARDIAN file system provides indications of a number of errors and other special conditions. These conditions may occur during execution of almost any user application or Tandem-supplied program, since most programs use the file system.

Each file system procedure sets the hardware condition code to indicate the outcome of the operation. The condition code should be checked immediately following each call to a file system procedure. Typically this is done as follows:

```
CALL READ (filenum,buffer,72,numxferred);
IF <> THEN...
```

If the "not equal" condition is detected, an error or warning condition occurred; the program should then check the error number and take appropriate action.

The error number associated with the last operation on a particular open file can be obtained by calling the FILEINFO procedure and passing the file number of the file in error:

```
CALL FILEINFO(in^file, err^num);
```

The error number associated with an operation on a file that is not open (such as a disc file creation or a file open failure) can be obtained by passing the file number as -1 to the FILEINFO procedure immediately following the call to the operation in question:

```
CALL FILEINFO(-1, err^num);
```

Note that the OPEN procedure returns -1 to <file number> if the open fails.

ERRORS

The GUARDIAN file system error numbers are grouped into four major categories:

Error	Category
0	No error. The operation executed successfully.
1-9	Warning. The operation executed with the exception of the indicated condition. For warning 6, data is returned in the application process's buffer.
10-255	Error. The operation encountered an error or a special condition which the application must recognize -- for instance, an aborted transaction on a TMF audited file.

If a device type number includes a dot (.), the digits to the left of the dot are the device type, and the digits to the right of the dot are the device subtype. Device subtypes are listed in table 2-3 (in the "File System Procedures" section, under the DEVICEINFO procedure).

The error explanations given here are intentionally general. The state of the system and the appropriate corrective action often depend on the application, the device, and/or the Tandem-supplied programs being used when the error occurred. For more information on errors related to terminals, line printers, tape drives, card readers, interprocess communication, and the operator console, refer to sections 2.5 through 2.10 of this manual. For additional information on errors related to Tandem subsystems such as ENSCRIBE, PATHWAY, TMF, ENVOY, or AXCESS, refer to the manuals on those subsystems.

In particular, file system errors returned by the data communication subsystems (device types 6, 7, 9, 10, 11, 12, 26, and 59 through 63) may have special meanings depending on the particular subsystem, access method, or protocol being used. If one of these device types is involved, refer also to the appropriate one of these manuals or sections:

- Section 2.5 of this manual (device type 6, Tandem model 6511, 6512, 6520 and 6530 terminals only)
- AXCESS Data Communications Programming Manual (device type 6 models other than listed above; also device types 9, 10, 59, 60, and 61)
- ENVOY Byte-Oriented Protocols Reference Manual (device type 7)
- ENVOYACP Bit-Oriented Protocols Reference Manual (device type 11)
- Tandem to IBM Link (TIL) Reference Manual for Tandem Users (device type 12)
- Tandem HyperLink Reference Manual (device type 26)
- EXPAND Users Manual (device types 62 and 63)

Refer to the ENSCRIBE Programming Manual for information on structured and unstructured disc files. For information on PUP and the operator console, refer to the Tandem NonStop System Operations Manual or the Tandem NonStop II System Operations Manual. For information on system configuration, refer to the Tandem NonStop System Management Manual or the Tandem NonStop II System Management Manual.

Note: Unless otherwise specified, all information applies to both NonStop systems and NonStop II systems. "I only" means the information applies only to NonStop systems; "II only" means it applies only to NonStop II systems.

5 (%5) FAILURE TO PROVIDE SEQUENTIAL BUFFERING (I only)
(device type: 3)
A structured disc file was opened for sequential block buffering, but the specified sequential block buffer length was not sufficient to contain a data block from the file. The open succeeds, but normal system buffering is used. Correct the program to satisfy criteria for sequential buffering.

6 (%6) SYSTEM MESSAGE RECEIVED (device type: 2)
The process received a system message from another process. This is generally not an error, but an indication that the message just read from $RECEIVE is a system message. Data is returned in the application process's buffer. Program action on receipt of a system message depends on the application.

7 (%7) PROCESS NOT ACCEPTING CONTROL, SETMODE, RESETSYNC OR CONTROLBUF MESSAGES (device type: 0)
CONTROL, SETMODE, RESETSYNC or CONTROLBUF was called for a process file, but the latter process did not open its $RECEIVE file with <flags>.<1> set to 1 to enable receipt of these messages. Open the process with <flags>.<1> set to 1, correct the file operand on the procedure call, or eliminate the call.

8 (%10) OPERATION SUCCESSFUL (EXAMINE MCW FOR ADDITIONAL STATUS)
(device type: 11.40, 11.42)
An operation to an ENVOYACP data communication line (SDLC or ADCCP) completed successfully, but additional status information was also received. Retrieve information from the Message Control Word (MCW) before proceeding.

CONDITION CODE < (CCL): ERRORS

10 (%12) FILE OR RECORD ALREADY EXISTS (device type: 3)
An operation requested creation of a new disc file, insertion of a new record in a key-sequenced file, or insertion of a new record with a unique alternate key in a structured disc file, but a file by that name or a record with that primary key already existed. Corrective action is application-dependent.

11 (%13) FILE NOT IN DIRECTORY OR RECORD NOT IN FILE
(device type: 3)
An operation referred to a nonexistent disc file or record. Corrective action is application-dependent; for "record not in file", it depends on the positioning mode.

12 (%14) FILE IN USE (device type: any except 2)
Specified file was being used, with exclusive or protected access, by another process. Corrective action is application-dependent. User processes can reply with this error if they have opened $RECEIVE to enable receipt of OPEN and CLOSE system messages (OPEN <flags>.<0> = 1) and with a receive depth greater than zero.

21 (%25) ILLEGAL COUNT SPECIFIED (device type: any except 2)
An illegal count parameter was specified in a file system call, or the operation attempted to transfer too much or too little data. For structured disc files, this may also occur for creation of a file with an invalid record length or alternate key length, or for a file access specifying an inconsistent key length or compare length. This is a coding error; corrective action is dependent on the device or the application.

22 (%26) APPLICATION PARAMETER OR BUFFER ADDRESS OUT OF BOUNDS (device type: any)
An out-of-bounds application parameter or buffer address parameter was specified in a file system call; that is, a pointer to the parameter or the buffer has an address which is greater than the MEM associated with the data area of the process. This is a coding error; corrective action is application-dependent.

23 (%27) DISC ADDRESS OUT OF BOUNDS (device type: 3)
A disc address specified in a file system call was too large or too small. This error generally indicates corrupt data or a corrupt alternate key file. Corrective action is application-dependent.

24 (%30) PRIVILEGED MODE REQUIRED FOR THIS OPERATION (device type: any)
A non-privileged user or process attempted to perform an operation requiring privileged mode. Have the system manager license the program file.

25 (%31) AWAITIO OR CANCEL ATTEMPTED ON WAIT FILE (device type: any)
AWAITIO or CANCEL was called for a file opened for wait i/o. Open the file for no-wait i/o, correct the file number, or make another correction appropriate to the application.

26 (%32) AWAITIO, CANCEL, OR CONTROL 22 ATTEMPTED ON A FILE WITH NO OUTSTANDING REQUESTS (device type: any)
AWAITIO, CANCEL, or CONTROL 22 was called, but no i/o requests were outstanding on the file. Corrective action is application-dependent.

27 (%33) WAIT OPERATION ATTEMPTED WHEN OUTSTANDING REQUESTS PENDING (device type: any)
A wait operation (that is, an operation that cannot be performed no-wait, such as SETMODE, POSITION, KEYPOSITION, or SETPARAM) was attempted on a file that was opened no-wait, and outstanding no-wait i/o requests were pending on that file. Corrective action is application-dependent.

33 (%41) I/O PROCESS UNABLE TO OBTAIN IOPOOL SPACE FOR I/O BUFFER, OR COUNT TOO LARGE FOR DEDICATED I/O BUFFER (I only)
I/O PROCESS UNABLE TO OBTAIN I/O SEGMENT SPACE (II only)
READ FROM UNSTRUCTURED DISC SPANS TOO MANY SECTORS (both)
(device type: any except 2)

Insufficient buffer space was available for the i/o process; the count parameter for a dedicated i/o buffer was too large (NonStop systems only); or a read from an unstructured disc file spanned more than eight sectors. For an insufficient space error, wait and try again; if the problem persists, check the system for processes that are using too much memory for i/o. For a "count too large" error, make the buffer smaller. If the error is due to reading too many sectors from disc, re-code the application to read fewer sectors.

34 (%42) UNABLE TO OBTAIN FILE SYSTEM CONTROL BLOCK (II only)
(device type: any)

All file system control blocks were in use, so the given operation could not be performed. Wait and try again. If the problem persists, check the system for processes that are using too many open files.

35 (%43) UNABLE TO OBTAIN I/O PROCESS CONTROL BLOCK (II only)
(device type: any except 2)

All i/o process control blocks were in use, so the given operation could not be performed. Wait and try again. If the problem persists, check the system for processes that are performing too many concurrent i/o operations.

36 (%44) UNABLE TO OBTAIN PHYSICAL MEMORY (II only)
(device type: any)

Insufficient physical memory was available to perform the specified operation. Wait and try again. If the problem persists, check the system for processes that are using too much memory.

37 (%45) UNABLE TO OBTAIN PHYSICAL MEMORY FOR I/O (II only)
(device type: any except 2)

Insufficient physical memory was available to perform the specified i/o operation. Wait and try again. If the problem persists, check the system for processes that are using too much memory for i/o.

38 (%46) OPERATION ATTEMPTED ON WRONG TYPE OF SYSTEM
(device type: any except 2)

A program running on a NonStop system specified an operation available only on NonStop II systems, or a program running on a NonStop II system specified an operation available only on NonStop systems. Recode the application to eliminate the illegal operation.

47 (%57) KEY NOT CONSISTENT WITH FILE DATA (device type: 3)
For some reason (such as no disc space available), the alternate key file could not be updated on an insert, update, or delete operation to the primary file. The data is inserted in (or deleted from) the primary file, so the alternate key file is no longer consistent with the primary file. Corrective action is application-dependent.

48 (%60) SECURITY VIOLATION, OR REMOTE PASSWORD ILLEGAL OR DOES NOT EXIST (device type: 3)
Specified operation (read, write, execute, or purge) was not permitted on the given disc file by the given user, because of the way the file was secured when created or because of an illegal or nonexistent password in an EXPAND network environment. Re-secure the file or re-code the application; if access is across a network, ensure that matching user ID's and remote passwords are established at both nodes.

49 (%61) ACCESS VIOLATION (device type: any except 2)
Specified type of access (read, write, or execute) was not permitted on the given file by the given process. This error may occur because that process did not open it for that kind of access, or because another process had it open in protected or exclusive mode. Corrective action depends on the application.

50 (%62) DIRECTORY ERROR (device type: 3)
A severe problem occurred with the directory on a disc volume. The file associated with the error is no longer accessible, although other files on the volume may be. It may be possible to recover some files by using the PUP REBUILDDFS command. Call your Tandem representative.

51 (%63) DIRECTORY IS BAD (device type: 3)
A severe problem occurred with the directory on a disc volume. The file associated with the error is no longer accessible, although other files on the volume may be. It may be possible to recover some files by using the PUP REBUILDDFS command. Call your Tandem representative.

52 (%64) ERROR IN DISC FREE SPACE TABLE (device type: 3)
A severe problem occurred on a disc volume. The file associated with the error is no longer accessible, although other files on the volume may be. It may be possible to recover some files by using the PUP REBUILDDFS command. Call your Tandem representative.

53 (%65) FILE SYSTEM INTERNAL ERROR (device type: 3)
A severe problem occurred on a disc volume used by the file system. The file associated with the error is no longer accessible, although other files on the volume may be. It may be possible to recover some files by using the PUP REBUILDDFS command. Call your Tandem representative.

60 (%74) VOLUME ON WHICH THIS FILE RESIDES HAS BEEN REMOVED, DEVICE HAS BEEN DOWNED, OR PROCESS HAS FAILED SINCE THE FILE WAS OPENED (device type: any except 1 and 2)

Specified file resided on a volume that was removed or a device that was downed since the file was opened; or for a process file, the specified process failed after the open but before this i/o operation. Ensure that the device (if any) is up, close the file and re-open it, then try again. In particular, this error is returned by a server written in COBOL, FORTRAN, or MUMPS if it receives a message from a process that it does not recognize as having previously opened it. The server maintains a table of processes from which it has received an OPEN message. Suppose a server named $X dies after a requestor opens it, and that another process also named $X is then created. The new $X knows nothing about the previous open from the requestor. If the requestor sends a message to "$X", the new $X receives it, then automatically replies with error 60 if it is written in COBOL, FORTRAN, or MUMPS.

61 (%75) NO MORE FILE OPENS PERMITTED ON THIS VOLUME (device type: 3)

The system operator had inhibited file opens on this volume by means of a PUP STOPOPENS command, or the number of open files on the volume had reached the maximum. Retry when opens are allowed.

62 (%76) VOLUME HAS BEEN MOUNTED, BUT MOUNT ORDER HAS NOT BEEN GIVEN (device type: 3)

Specified disc volume was physically mounted, but the mount order had not yet been given; the file could not be opened. Retry after the mount completes.

63 (%77) VOLUME HAS BEEN MOUNTED AND MOUNT IS IN PROGRESS (WAITING FOR MOUNT INTERRUPT) (device type: 3)

Specified disc volume was physically mounted and the mount order had been given, but the mount had not completed; the file could not be opened. Retry after the mount completes.

64 (%100) VOLUME HAS BEEN MOUNTED AND MOUNT IS IN PROGRESS (device type: 3)

Specified disc volume was physically mounted and the mount order had been given, but the mount had not completed; the file could not be opened. Retry after the mount completes.

65 (%101) ONLY SPECIAL REQUESTS PERMITTED (device type: 3)

Specified disc volume was upped in special request mode by the system operator. Retry after the volume is upped in normal mode.

76 (%114) TRANSACTION IS IN THE PROCESS OF ENDING
(device type: 3 or none)
For systems with TMF, the transaction was ending, so it could not be aborted or resumed. See the TMF Users Guide.

77 (%115) A TMF SYSTEM FILE HAS THE WRONG FILE CODE
(device type: 3)
For systems with TMF, a serious error occurred with a TMF system file. The cause of this error may be a user program that has corrupted a TMF system file, a catastrophic system failure, or an internal software error. Call your Tandem representative.

78 (%116) TRANSID IS INVALID OR OBSOLETE
(device type: 3 or none)
For systems with TMF, the transaction was invalid or obsolete. See the TMF Users Guide.

79 (%117) ATTEMPT MADE BY TRANSID TO UPDATE OR DELETE A RECORD IT HAS NOT PREVIOUSLY LOCKED
(device type: 3)
For systems with TMF, the transaction failed to lock a record before attempting to change or delete it. See the TMF Users Guide.

80 (%120) INVALID OPERATION ATTEMPTED ON AUDITED FILE OR NON-AUDITED DISC VOLUME (device type: 3)
For systems with TMF, an invalid operation was attempted on an audited file or a non-audited disc volume. See the TMF Users Guide.

81 (%121) ATTEMPTED OPERATION INVALID FOR TRANSID THAT HAS NO-WAIT I/O OUTSTANDING ON A DISC OR PROCESS FILE
(device type: 2 or none)
For systems with TMF, the attempted operation was invalid because the transaction had one or more outstanding no-wait i/o operations on a disc or process file. See the TMF Users Guide.

82 (%122) TMF IS NOT RUNNING (device type: 0, 3, or none)
BEGINTRANSACTION failed because TMF was not running on this system; or an i/o operation to an audited disc or a process file on a remote system was part of a TMF transaction, but TMF was not running on the remote system. Ensure that TMF is running on all systems involved in the transaction.

83 (%123) PROCESS HAS INITIATED MORE CONCURRENT TRANSACTIONS THAN CAN BE HANDLED (device type: none)
For systems with TMF, BEGINTRANSACTION failed because the process had reached its maximum number of concurrent transactions -- a number equal to TFILE-depth, or 1 if the TFILE was not open. See the TMF Users Guide.

98 (%142) TRANSACTION MONITOR PROCESS'S NETWORK ACTIVE TRANSACTIONS TABLE IS FULL (device type: 0, 3, or none)
For systems with TMF, BEGINTRANSACTION failed because the TMF Network Active Transactions Table on this system was full; or an i/o operation to an audited disc or a process file on a remote system was part of a TMF transaction, but the TMF Network Active Transactions Table on the remote system was full. See the TMF Users Guide.

99 (%143) ATTEMPT TO USE MICROCODE OPTION THAT IS NOT INSTALLED (device type: any except 2)
Attempt was made to use features in a microcode option that was not installed in the system (such as ENSCRIBE structured disc files on a NonStop system). Ensure that the system has the required microcode.

100 (%144) DEVICE NOT READY (device type: any except 2)
Device was not powered up, not on line, or (for a card reader) out of cards. Make device ready.

101 (%145) NO WRITE RING (device type: 4)
Mounted tape could not be written to because it had no write ring. Remove tape, put a write ring on it, remount tape and try again.

102 (%146) PAPER OUT OR BAIL NOT PROPERLY CLOSED (device type: 5)
Printer could not continue because it was out of paper, or because the paper bail was not in place. Load more paper or close the bail.

103 (%147) DISC NOT READY DUE TO POWER FAILURE (device type: 3)
Disc device was not ready because of a system power failure. Wait and try again.

104 (%150) NO RESPONSE FROM DEVICE (device type: 5.4)
Printer did not return the requested status; either the printer power was off or a hardware problem occurred. Power up the device or repair it.

105 (%151) VFU ERROR (device type: 5.4)
The printer DAVFU buffer was invalid. This can occur for the following reasons: 1) more than one stop was defined for channel 0 (top of form); 2) no stops were defined for one or more channels; 3) bits 12 through 15 of each word were not zeros. Correct the programming error.

110 (%156) ONLY BREAK ACCESS PERMITTED (device type: 6 or 61)
Specified terminal could not be accessed because BREAK was typed and break mode had been specified when BREAK was enabled. No data was transferred. The terminal is inaccessible (unless this process uses SETMODE to signal its operations as break access) until the process processing the break calls SETMODE function 12 to allow normal access to the terminal. If this process is not the one that enabled BREAK, retry the operation periodically. If this process enabled BREAK, check $RECEIVE for the system BREAK message and take appropriate action.

130 (%202) ILLEGAL ADDRESS TO DISC (device type: 3)
The requested address was too large for disc, or (for disc subtypes >= 5 on NonStop systems or any disc subtype on NonStop II systems) an error occurred while the disc was being formatted. Correct the coding error or reformat the disc.

131 (%203) WRITE CHECK ERROR FROM DISC (device type: 3.0 or 3.1)
An internal circuitry fault was detected by the disc hardware. Call your Tandem representative.

132 (%204) SEEK INCOMPLETE FROM DISC (device type: 3.0 or 3.1)
The disc read/write heads did not reach the desired cylinder address after a retry. Call your Tandem representative.

133 (%205) ACCESS NOT READY ON DISC (device type: 3.0 or 3.1)
The disc read/write heads did not reach the desired cylinder address. Call your Tandem representative.

134 (%206) ADDRESS COMPARE ERROR ON DISC (device type: 3)
A header search failure or header miscompare occurred on the disc. This indicates either a request for a bad address or, possibly, a head alignment or formatting problem. Call your Tandem representative.

135 (%207) WRITE PROTECT VIOLATION WITH DISC (device type: 3)
An attempt was made to write to a write-protected disc. Reset the write-protect switch to allow writes.

136 (%210) UNIT OWNERSHIP ERROR (DUAL-PORT DISC) (device type: 3)
A hard error occurred in the disc port logic. Call your Tandem representative.

137 (%211) CONTROLLER BUFFER PARITY ERROR
(device type: any except 2)
A parity error occurred in the controller buffer. Call your Tandem representative.

138 (%212) INTERRUPT OVERRUN (device type: any except 2)
Device interrupted the processor more quickly than the software could respond. Wait and try again. If retries do not recover from this error, call your Tandem representative.

139 (%213) CONTROLLER ERROR (device type: any except 2)
Controller failed its internal diagnostics. Call your Tandem representative for a replacement.

140 (%214) MODEM ERROR, OR MODEM OR LINK DISCONNECTED
(device type: 6, 7, 10, 11, 12, 59, 60, 61, or 63)
A modem error occurred; for instance, the communication link was not yet established, a modem failure occurred, a momentary loss of carrier occurred, or the modem or link was disconnected. Corrective action is device-dependent.

155 (%233) ONLY NINE-TRACK TAPE PERMITTED (device type: 4)
A seven-track tape device was specified for an operation requiring nine-track tape (most Tandem subsystems require nine-track). Retry, specifying a drive configured for nine-track tape.

156 (%234) TIL PROTOCOL VIOLATION DETECTED (device type: 12)
Either a connect request (CONTROL 11) was made after the TIL link had already been connected, or an internal link error occurred. The call is aborted and the link is disconnected. In the first case, correct the coding error; in the second, call your Tandem representative.

157 (%235) I/O PROCESS INTERNAL ERROR
(device type: any except 2)
An internal system error occurred. Call your Tandem representative.

158 (%236) INVALID FUNCTION REQUESTED FOR HYPERLINK
(device type: 26)
Operation specified an invalid Tandem HyperLink function code. Supply the correct function code.

160 (%240) REQUEST IS INVALID FOR LINE STATE
(device type: 6, 7, 10, or 11)
MORE THAN 7 READS OR 7 WRITES ISSUED (device type: 11)
A protocol error occurred. Corrective action is device-dependent.

161 (%241) IMPOSSIBLE EVENT OCCURRED FOR LINE STATE
(device type: 7, 10, or 11)
An event occurred that was impossible for the current line state; this probably indicates a hardware problem. Corrective action is device-dependent.

162 (%242) OPERATION TIMED OUT (device type: 7, 10, or 11)
Specified operation timed out after several retries. Corrective action is device-dependent.

163 (%243) EOT RECEIVED (device type: 7.0, 7.1, 7.2, 7.3, or 7.8)
POWER AT AUTO-CALL UNIT IS OFF (device type: 7.56 or 11)
An EOT was received while waiting for a line bid or for a message, or the power at the auto-call unit was off. Corrective action is device-dependent.

164 (%244) DISCONNECT RECEIVED
(device type: 7.0, 7.1, 10, 11, or 61)
DATA LINE IS OCCUPIED (BUSY) (device type: 7.56 or 11)
A disconnect was received or a send disconnect call was issued while a request was outstanding, or the data line was busy. Corrective action is device-dependent.

165 (%245) RVI RECEIVED (device type: 7.0, 7.1, 7.2, or 7.3)
DATA LINE NOT OCCUPIED AFTER SETTING CALL REQUEST
(device type: 7.56 or 11)
An RVI was received, or the data line was not occupied after setting the call request. Corrective action is device-dependent.

173 (%255) MAXIMUM ALLOWABLE NAKS RECEIVED (TRANSMISSION ERROR)
(device type: 6, 7, or 10)
INVALID MCW ON WRITE (device type: 11)
Specific meaning and corrective action for this error are device-dependent. For a 6520, 6524 or 6530 terminal, check power and turn on if necessary.

174 (%256) WACK RECEIVED AFTER SELECT (device type: 7.2 or 7.3)
ABORTED TRANSMITTED FRAME (device type: 11)
A WACK sequence was received as the text acknowledgment; or a link request occurred while the request was pending, possibly causing loss of data. Corrective action is device-dependent.

175 (%257) INCORRECT ALTERNATING ACK RECEIVED
(device type: 7.0, 7.1, 7.2, or 7.3)
COMMAND REJECT (device type: 11)
An incorrect alternating ACK was received, or a command reject condition was generated. Corrective action is device-dependent.

176 (%260) POLL SEQUENCE ENDED WITH NO RESPONDER
(device type: 7.3, 7.8, 7.9, or 11.40)
The poll sequence ended, but no message was received in response. Corrective action is device-dependent.

177 (%261) TEXT OVERRUN (device type: 7, 10, or 11)
Data received on a read exceeds the amount allowed by the read count. Corrective action usually involves increasing the read count; refer to the manual for the device for more information.

178 (%262) NO ADDRESS LIST SPECIFIED
(device type: 7.2, 7.3, 7.8, 7.9, 11.40, or 61)
An address list was required for this operation, but none was specified. Corrective action depends on the device and the application.

179 (%263) APPLICATION BUFFER IS INCORRECT (device type: 10 or 61)
CONTROL REQUEST PENDING OR AUTOPOLL ACTIVE
(device type: 11.40)
For an AXCESS communication line, an error was encountered in the application buffer; for an ENVOYACP line, the operation could not be performed because a control request was pending or the auto-poll feature was active. Corrective action depends on the device and the application.

180 (%264) UNKNOWN DEVICE STATUS RECEIVED
(device type: 5.3, 6.6 through 6.10, or 10)
An invalid device status was received and could not be translated into a usable error number. Use CUP to determine the status received.

181 (%265) STATUS RECEIPT CURRENTLY ENABLED FOR SUBDEVICE
(device type: 10)
Data was sent to the subdevice when it expected to receive status information. Corrective action is application-dependent.

201 (%311) CURRENT PATH TO THE DEVICE IS DOWN
(device type: any except 0 and 2)
ATTEMPT WAS MADE TO WRITE TO A NONEXISTENT PROCESS
(device type: 0)

For a process file, an attempt was made to write to a nonexistent process, or a pending WRITE or WRITEREAD was aborted because the server process read the request using READUPDATE but died before it replied. For a device, the current path to the device was down. Either the operation never got started, or the operation completed but the path failed before a reply could be made. For a process file opened with a sync depth of zero, if the process file is a process pair, the primary process failed; retry the operation once to cause communication with the backup process. If a second error 201 occurs, no backup exists (both paths are down), and programmatic recovery is impossible. (For disc server processes, the latter error recovery scheme may be risky; to avoid problems, open these processes with a sync depth greater than zero.) Take corrective action appropriate to the device and the application.

210 (%322) DEVICE OWNERSHIP CHANGED DURING OPERATION
(device type: any except 2)

A path switch to a hardware device controller occurred while this operation was in progress. This error is associated with concurrent operations involving more than one unit connected to a multi-unit controller. It occurs when an operation is in progress with one unit on a multi-unit controller and an error is detected during an operation with another unit on the same controller. (The other operation could have been on behalf of this or another application process.) The associated medium might have moved. Corrective action depends on the device and the application.

211 (%323) FAILURE OF CPU PERFORMING THIS OPERATION
(device type: any)

The processor module controlling the device associated with this file operation failed (path error). The file operation itself stopped at some indeterminate point, and the associated medium might have moved. Corrective action depends on the device and the application.

212 (%324) EIO INSTRUCTION FAILURE (I only)
(device type: any except 2)

A controller failure occurred (path error). The file operation stopped at some indeterminate point, and the associated medium might have moved. Corrective action depends on the device and the application.

213 (%325) CHANNEL DATA PARITY ERROR
(device type: any except 2)

A controller or channel failure occurred (path error). The file operation stopped at some indeterminate point, and the associated medium might have moved. Corrective action depends on the device and the application.

221 (%335) CHANNEL PAD-IN VIOLATION (I only)
CONTROLLER HANDSHAKE VIOLATION (II only)
(device type: any except 2)
A controller or channel failure occurred (path error). The file operation stopped at some indeterminate point, and the associated medium might have moved. Corrective action depends on the device and the application.

222 (%336) BAD CHANNEL STATUS FROM EIO INSTRUCTION
(device type: any except 2)
A controller or channel failure occurred (path error). The file operation stopped at some indeterminate point, and the associated medium might have moved. Corrective action depends on the device and the application.

223 (%337) BAD CHANNEL STATUS FROM IIO INSTRUCTION
(device type: any except 2)
A controller or channel failure occurred (path error). The file operation stopped at some indeterminate point, and the associated medium might have moved. Corrective action depends on the device and the application.

224 (%340) CONTROLLER ERROR (I only)
(device type: any except 2)
A controller failure occurred (path error). The file operation stopped at some indeterminate point, and the associated medium might have moved. Corrective action depends on the device and the application.

225 (%341) NO UNIT OR MULTIPLE UNITS ASSIGNED TO SAME UNIT NUMBER
(device type: any except 2)
A path error occurred because no unit or multiple units were assigned to the unit number being used. The file operation stopped at some indeterminate point, and the associated medium might have moved. Corrective action depends on the device and the application. If the device is a disc, check the UNITS plugs on the drive.

226 (%342) CONTROLLER BUSY ERROR (I only)
(device type: any except 2)
A controller failure occurred (path error). The file operation stopped at some indeterminate point, and the associated medium might have moved. Corrective action depends on the device and the application.

230 (%346) CPU POWER ON DURING OPERATION
(device type: any except 2)
CPU power failed, then was restored during this operation (path error). At least one path, and possibly both paths, were operable. The file operation stopped at some indeterminate point, and the associated medium might have moved. Corrective action depends on the device and the application.

ERROR RECOVERY

The programmer must consider a number of items when writing error recovery routines:

- The type of device (e.g., disc, magnetic tape, line printer, etc.)
- The type of error (i.e., whether it is recoverable programmatically)
- If not a disc, whether or not medium movement took place
- The number of no-wait operations outstanding when the error was detected

Device

Disc - For disc files, if a file is opened with a sync depth greater than or equal to 1, all recoverable errors are automatically retried by the file system.

Terminals - Usually, errors occurring when writing to a terminal can simply be retried. Some errors occurring when reading, however, indicate a failure at some indeterminate point. In those cases, a message should be sent to the terminal operator indicating that the last entry should be retyped before retrying the read.

Printers - The medium may or may not have moved when certain errors occurred. Simply reprinting a line associated with an error could result in lines being duplicated or, if no-wait i/o is used, printed out of order.

Magnetic Tapes - As with printers, the medium (tape) may or may not have moved when certain errors occurred. For error recovery, the application program should keep track of the number of records read or written. In some cases it may be necessary for the application program to rewind the tape, then space forward to the record prior to where an error occurred.

Interprocess - For the $RECEIVE file, there are no recoverable errors. For WRITE or WRITEREAD to a process, if the file is opened with a sync depth greater than zero, all recoverable errors are automatically retried by the file system.

Operator Console - All recoverable errors are automatically retried by the file system.

Path Errors (Errors 200-255)

Figure 2-16 illustrates a path failure, the error returned to the application process, the retry by the application process, and the successful outcome of the retry.

3. For disc files, and also for process files when the process or its cpu fails, the file system has the capability to recover automatically from path failures. This capability is in effect if a sync depth greater than zero is specified when the file is opened. For a more detailed explanation of this feature, see "Automatic Path Error Recovery for Disc Files" in section 2.1.

   Note that the file system retries at most one time per operation. This one-retry limit is not a real concern for disc files, because cpu failures are infrequent and disc process requests are usually processed very quickly. However, an i/o request to a server process may be outstanding for a long time, perhaps several hours. If the primary server process dies, the file system redirects the WRITE or WRITEREAD operation to the backup server process. But even if the backup re-creates a new backup, the file system will not retry the operation a second time if the new primary also fails.

Table 2-6 summarizes path error recovery for procedures applicable to non-disc devices.

Table 2-6. Path Error Recovery for Devices Other than Discs and Processes

Procedure	Path Error Recovery Performed by:		Path Error Not Possible
	System	Application	
AWAITIO		x	
CANCELREQ			x
CLOSE	x		
CONTROL		x	
CONTROLBUF		x	
FILEINFO	x		
OPEN	x		
READ		x	
SETMODE	x		
SETMODENOWAIT	x		
WRITE		x	
WRITEREAD		x	
WRITEUPDATE		x	

For those procedures listed under "System", if an error 200 or greater is returned, programmatic error recovery is not possible.

For those procedures listed under "Application", the application program must perform path error recovery. The application program should keep track of the number of times any of the path errors from

- I/O Messages

  These messages log transient and hard (i.e., unrecoverable) i/o failures, log the up/down state of i/o devices, and log device statistical information.

- Resource Allocation Messages

  These messages log transient and hard resource allocation failures, and indicate potential resource allocation problems and unusual resource allocation conditions. The principal reason for these messages is twofold: to aid in the diagnosis of configuration errors in the areas of Link Control Block (LCB) allocation and resident memory pool allocation, and to help pinpoint application processes that are being allocated an unusual number of LCB's and therefore are potentially detrimental to system operation.

Both types of console messages are described in the NonStop System Operations Manual and the NonStop II System Operations Manual.

The GUARDIAN file system can communicate with virtually any conversational mode or page mode terminal whose characteristics can be defined through the system generation program (SYSGEN).

The file system provides for data transfers between application processes and terminals in blocks of 0 to 4,095 bytes.

Topics covered in this section are:

- General Characteristics of Terminals
- Summary of Applicable Procedures
- Accessing Terminals
  - Transfer Termination when Reading
  - Transfer Modes
- Transparency Mode
- Checksum Processing
- Echo
- Timeouts
- Modems
- Break Feature
- Error Recovery
- Configuration Parameters
- Summary of Terminal CONTROL and SETMODE Operations

GENERAL CHARACTERISTICS OF TERMINALS

- Terminals are accessed by

  $<device name> or

  $<logical device number>.

- The maximum number of concurrent opens permitted a given terminal is eight (8).

- The logical device number of the home terminal where an application process was created can be obtained through the MYTERM utility procedure.

- Conversational mode/page mode terminal device type is 6.

- The asynchronous terminal multiplexer hardware has the capability to examine each character received from a terminal and compare the characters with four programmable "interrupt" characters. These characters are called interrupt characters because the receipt of one of these characters by the terminal multiplexer causes a hardware i/o interrupt to occur (the interrupt is invisible to application processes). The interrupt results in the system i/o process controlling the terminal being notified of the character's reception. Action appropriate for the particular interrupt character is then taken (in some cases this means notifying the application process).

  - Conversational mode line termination character
  - Conversational mode backspace type
  - Conversational mode CR/LF delay
  - Conversational mode forms control delay
  - Page mode page termination character
  - Page mode pseudo-polling trigger character

- Tandem-supplied programs using terminals open them with share access.

- Default file system spacing mode is "post-space" (i.e., space after printing). The spacing mode can be set to "pre-space" (i.e., space before printing) via a SETMODE function.

SUMMARY OF APPLICABLE PROCEDURES

The following procedures are used to perform input/output operations with terminals:

DEVICEINFO	provides device type and configured record length
OPEN	establishes communication with a file
READ	reads information from an open file
WRITE	writes information to an open file
WRITEREAD	writes, then waits for data to be read back from an open terminal
CONTROL	is used for forms control and modem connect/disconnect
AWAITIO	waits for completion of an outstanding i/o operation pending on an open file
CANCELREQ	cancels the oldest outstanding operation identified by a tag on an open file
FILEINFO	provides error information and characteristics about an open file
SETMODE	sets/clears the following functions:

- single spacing
- conversational/page mode
- parity checking
- access mode
- read termination on interrupt character
- baud rate
- system parity generation
- reset to default values
- auto line feed
- interrupt characters
- break ownership
- read termination on ETX
- echo
- character size
- spacing mode (pre/post-spacing)

only one buffer, "term^buffer", is specified; the data is returned there):

```
.
term^buffer := ": ";                    ! prompt.
.
CALL WRITEREAD ( home^term^num, term^buffer, 1, 72, num^read );
.
```

Writes ":" on the terminal then waits for input.

Note: WRITEREAD does not issue a carriage return/line feed character sequence to the terminal after the write phase of the write/read sequence.

The WRITEREAD procedure is also useful for issuing control commands to a terminal. For example, to read a seven-character cursor address from a terminal that requires a control character sequence of "ESC, a, DC1" (escape character followed by lower-case "a" character, followed by a device control 1 character), the following could be written in an application program:

```
.
term^buffer ':=' [%015541, % 010400 ]; ! "ESC a DC1".
.
CALL WRITEREAD ( home^term^num, term^buffer, 3, 7, num^read );
.
```

After the WRITEREAD completes, "term^buffer" contains the cursor address and seven is returned to "num^read".

Transfer Termination when Reading

A READ or WRITEREAD from a terminal is terminated when any of the following conditions is encountered:

- Interrupt character checking is enabled and a line termination character is input from a conversational mode terminal (see "Line Termination Character"). On return from READ or WRITEREAD, <buffer> contains <count read> characters, and the condition code indicator is set to CCE. The receipt of the line termination character is not reflected in the <count read> value.

- Interrupt character checking is enabled and an EOF character is input from a conversational mode terminal. On return from READ or WRITEREAD, nothing is transferred into <buffer>, <count read> = 0, and the condition code indicator is set to CCG.

- Interrupt character checking is enabled and an application-defined interrupt character is input. If the application-defined character differs from the system-defined interrupt characters, then on the return from READ or WRITEREAD, <buffer> contains <count read> characters, the last character being the interrupt character, and the condition code indicator is set to CCE.

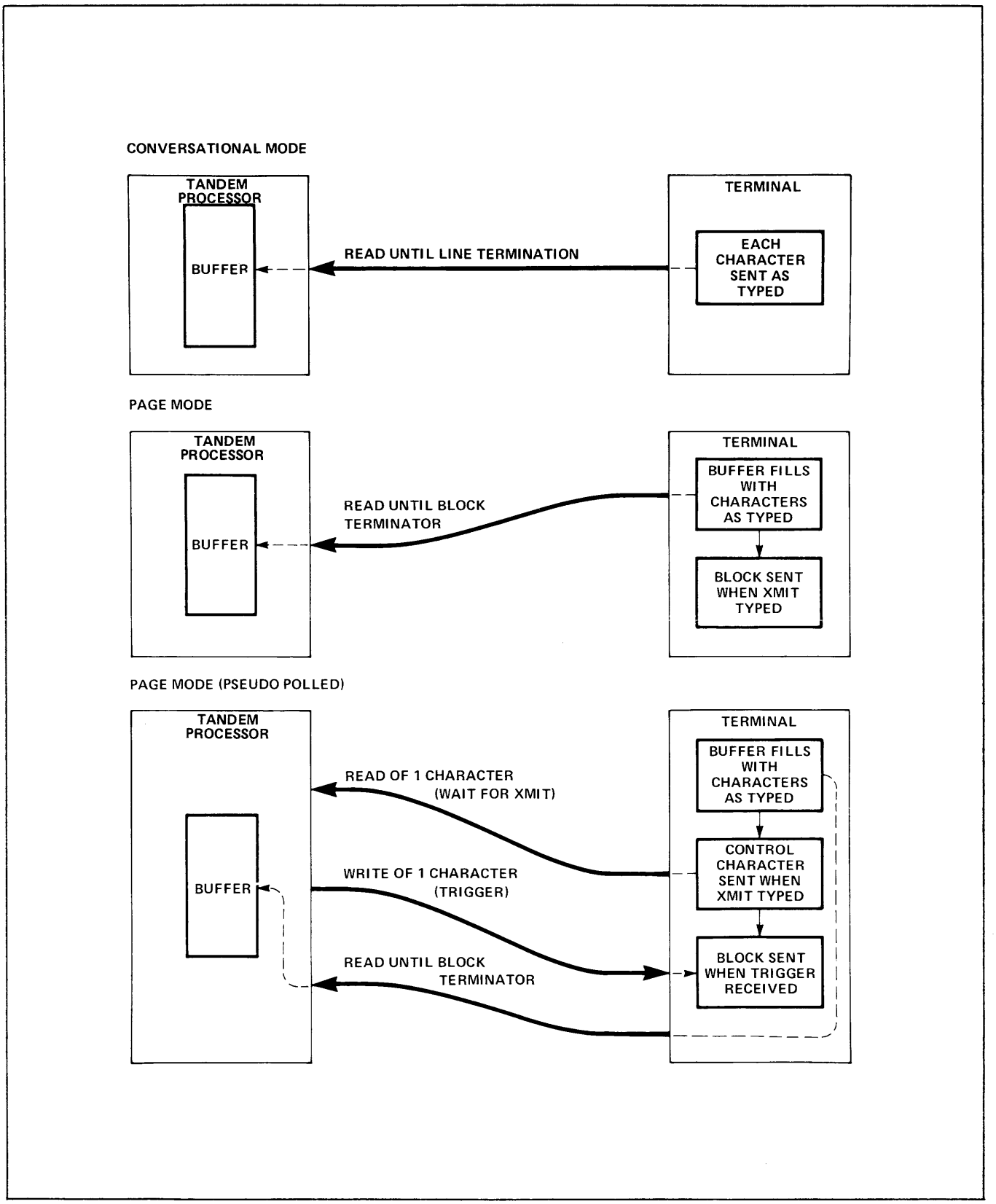


Figure 2-17. Transfer Modes for Terminals

LINE TERMINATION CHARACTER. The line termination character, when received from a terminal, signals the computer system that the current line transfer is completed. Line termination characters for each conversational mode terminal connected to the system are specified (i.e., configured) at system generation time.

There are special characteristics associated with receiving the line termination character:

- It is not counted in the <count read> returned from the READ or WRITEREAD procedures, although it is transferred into the application's buffer if an odd-byte-count read is executed.

- If carriage return (%015) is the configured line termination character, another device configuration parameter specifies whether or not the file system should provide automatic line spacing on the terminal. This is done by automatically issuing a line feed character (%012) to the terminal after receiving the carriage return character. (Typically, the line feed character is issued if the terminal does not provide its own line feed.)

  Automatic issuance of the line feed character can be changed programmatically through SETMODE function 7.

- If any character other than carriage return is the configured line termination character, the file system always issues a carriage return/line feed sequence to the terminal.

- The line is terminated automatically when the number of characters specified in the <read count> parameter are input. If termination on <read count> occurs, the file system does not issue a carriage return/ line feed sequence to the terminal.

Here are some examples.

Carriage return is the configured line termination character, and a read of 72 characters is issued to a terminal:

```
.
CALL READ ( home^term^num, buffer, 72, num^read );
.
```

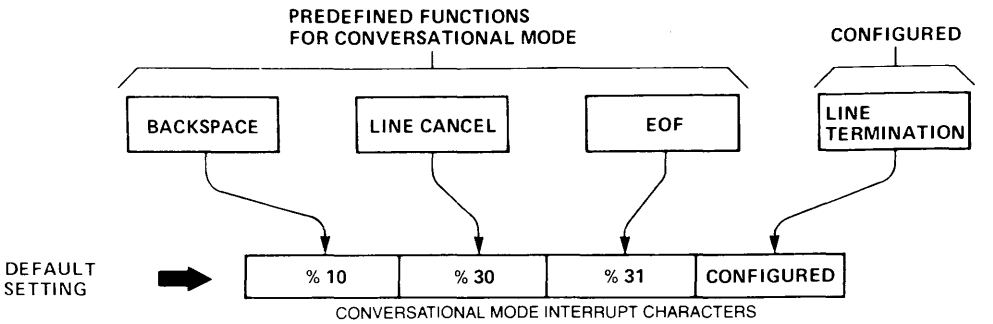
Then the terminal operator types in the following information:

```
NOW IS THE TIME<cr>
```

- initial cursor position

"NOW IS THE TIME" is returned in "buffer", 15 is returned in "num^read", and the file system issues a line feed to "home^term^num".

If, instead, operator just typed in a carriage return:



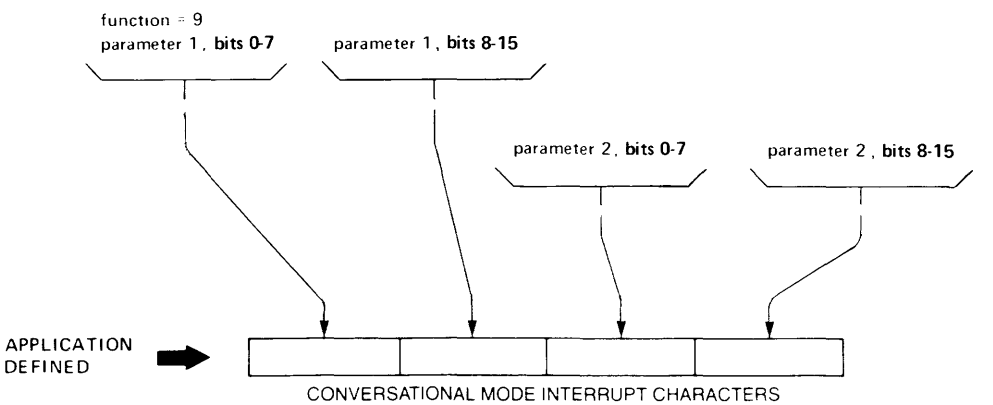
ACTION OF PREDEFINED FUNCTIONS AND LINE TERMINATION:

- BACKSPACE: DELETE LAST CHARACTER (BACKSPACE CURSOR)
- LINE CANCEL: DELETE LINE (@ CR LF ON TERMINAL)

} INVISIBLE TO APPLICATION PROGRAM

- END-OF-FILE: CCG, error = 1, AND count read = 0 (EOF! CR LF ON TERMINAL) –COMPLETES TRANSFER — TO APPLICATION
- LINE TERMINATION:
  - COMPLETES TRANSFER
  - LINE TERMINATION CHARACTER IS NOT TRANSFERRED INTO APPLICATIONS BUFFER
  - CR LF ON TERMINAL IF LINE TERMINATION NOT CARRIAGE RETURN (% 15)
  - LF ON TERMINAL IF LINE TERMINATION IS CARRIAGE RETURN AND AUTO LINE FEED IS ON

TO CHANGE INTERRUPT CHARACTERS:
CALL SET MODE (file number, function, parameter 1, parameter 2, last function);



ACTION WHEN NOT PREDEFINED OR LINE TERMINATION:

- COMPLETES TRANSFER
- INTERRUPT CHARACTER IS TRANSFERRED INTO BUFFER

Figure 2-18. Conversational Mode Interrupt Characters

```
.
CALL SETMODE ( home^term^num, set^sig, iintchars, iintchars[1] );
.
```

Then a read is issued to the terminal:

```
.
CALL READ ( home^term^num, buffer, 72, num^read );
.
```

The terminal operator enters the following information:

```
TODAY IS THE DAY<cntl-I>              ! <cntl-I> is the horizontal
^                                     ! tab character.
| - initial cursor position
```

"TODAY IS THE DAY<cntl-I>" is returned in <buffer>, 17 is returned in <num^read>. No line feed occurs on the terminal.

Next, the application checks the last character received to determine if, in fact, a <cntl-I> was entered:

```
.
IF buffer[num^read - 1] = %011 THEN.. ! horizontal tab.
.
```

Assuming that the application needed to move the cursor (indicating tabulation had occurred) to column 30, a call to SETMODE is issued to turn off single spacing, then a call to WRITE is issued to write blanks (%040) to the terminal:

```
.
CALL SETMODE ( home^term^num, set^space, no^space ); ! no spacing.
.
CALL WRITE ( home^term^num, blanks, 30-num^read, num^written );
.
```

After the write, the information on the terminal appears as:

```
TODAY IS THE DAY_____________
                ^
                | - cursor position
```

Then another read is issued to the terminal. This time the operator enters:

```
                FOR BEGINNING<cr>
                ^
                | - cursor position
```

"FOR BEGINNING" is returned in "buffer" (writing over the previous contents) and 13 is returned in "num^read". <cr> is not transferred into "buffer" or reflected in "num^read", because it is the line termination character.

```
WHAT EVER HAPPENED TO
↑
|  - initial and final cursor position
```

Because the application program, rather than the file system, is supplying the carriage return character, a delay (dependent on the particular terminal involved) may be needed to give the terminal ample time to perform the carriage return operation. This can be accomplished by writing a number of null characters to the terminal or calling the DELAY utility procedure (if no-wait i/o is used, the null character method must be used).

CONTROL operation 1 can be used to cause a form feed or vertical tabulation to occur on a terminal (provided, of course, that the terminal has the capability). The CONTROL <parameter> values for these operations are

```
0            =  form feed
1 or greater = vertical tab
```

For example, to cause a top-of-form advance on a hard-copy terminal, the following call to CONTROL is written in the application program:

```
  .
LITERAL forms^cont = 1,
        form^feed  = 0;
  .
CALL CONTROL ( home^term^num, forms^cont, form^feed );
  .
```

The file system automatically delays subsequent access to the same terminal for a configured period of time after performing forms control through the CONTROL procedure.

If the configured delay is not suitable, the application program can issue a form feed (%014) or vertical tabulation (%013) character through a WRITE procedure. However, in this case, a delay must be included in the application program to permit the actual forms movement to complete:

```
  .
DEFINE two^seconds = 200D#;
INT form^feed := %014 '<<' 8;
  .
  .
CALL WRITE ( home^term^num, form^feed, 1, num^written );
CALL DELAY ( two^seconds );
  .
```

The application process is suspended for two seconds after the form feed character is issued to the terminal.

- The <count read> parameter includes the interrupt character.

Note that special application-dependent interrupt characters can be mixed with the page termination character.

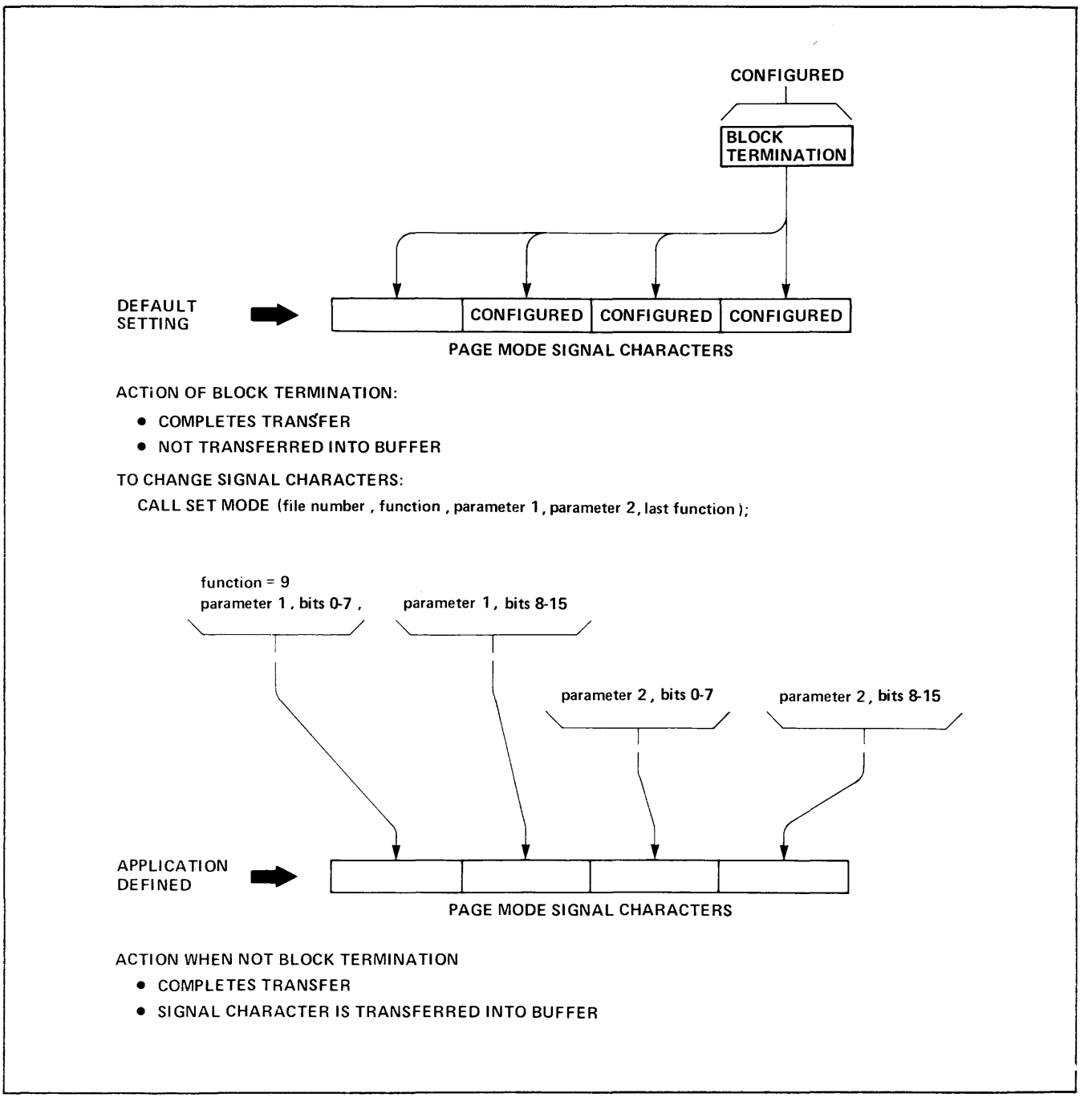


Figure 2-19. Page Mode Interrupt Characters

Then the file system transfer mode for the terminal is returned to conversational mode:

```
.
CALL SETMODE ( home^term^num, change^mode, conv^mode );
.
```

When returning to conversational mode, the interrupt characters are restored by the file system to their initial values: backspace, line cancel, end of file, and carriage return.

Note: When changing the interrupt characters through SETMODE, all four characters must be specified. Therefore, if less than four are needed, some character(s) must be duplicated.

PSEUDO-POLLED TERMINALS. During system generation, each pseudo-polled terminal is configured as to whether or not the file system automatically issues the trigger character when reading. To indicate automatic triggering, the actual trigger character is specified. To indicate that the application program will handle triggering, a trigger character with a value of zero (null) is specified.

The advantage of having the file system handle the triggering, of course, is that the operation is invisible to the application program. The automatic triggering only applies, however, when issuing a READ (not a WRITEREAD) to the terminal. WRITEREAD can still be used for such things as cursor sensing.

The advantage to having the application program handle the triggering is that practically no system buffer space (just one word) is used while the terminal operator is actually typing in information. The buffer space is allocated after the operator types the ENTER (or equivalent) key. (Terminals operating in normal page mode require that the entire system buffer space be allocated while waiting for a transfer to take place.)

Here's how it works:

The application program issues a read of one character to the pseudo-polled terminal (this read waits for the ready character):

```
.
CALL READ ( home^term^num, buffer, 1, num^read );
.
```

Reading one byte causes one word of system buffer space to be allocated.

The terminal operator types in the page of text, then types the ENTER key. Typing ENTER causes a ready character (e.g., device-control-2) to be sent to the computer system (causing the READ to complete).

```
  conv^mode     =     0, ! set mode param 1, set conversational mode.
  page^mode     =     1; ! set mode param 1, set page mode.

    .
! put the file into page mode.
CALL SETMODE ( term^num, change^mode, page^mode );
IF < THEN ... ; ! error.
```

sets the file system transfer mode for the terminal to "page mode".

```
! clear the screen, put the terminal into block mode, then wait
! for function key typed.
scontrol^buf ':=' [ esc, clear^spaces, esc, tblock^mode ];
CALL WRITEREAD ( term^num, control^buf, 4, 2 );
IF = THEN       ! function key typed.
  BEGIN
    .
      The returned data in the buffer can be examined to determine
      which function key was typed.
    .
  END
ELSE
IF < THEN ... ; ! error.
```

This call to WRITEREAD clears the terminal screen and puts the terminal into "block mode" (ESC ";", ESC "B"). The call to WRITEREAD completes when a function key is typed.

```
! read the screen.
sbuffer ':=' [ esc, sendall ];
CALL WRITEREAD ( term^num, buffer, 2, readcount , count^read );
IF < THEN ... ; ! error.
```

This call to WRITEREAD transfers a "send page unprotected" escape sequence to the terminal. The terminal responds by sending the screen from the home position to the previous cursor position. The WRITEREAD completes with the screen data (and field and line separator control characters) in "buffer" and "count^read" containing a count of all characters returned by the terminal.

```
! put the file back into conversational mode.
CALL SETMODE ( term^num, change^mode, conv^mode );
IF < THEN ... ; ! error.
```

sets the file system transfer mode for the terminal to "conversational mode".

```
! put the terminal into conversational mode.
scontrol^buf ':=' [ esc, tconv^mode ];
CALL WRITE ( term^num, control^buf, 2 );
IF < THEN ... ; ! error.
```

TIMEOUTS

Operations with terminals require human response, and therefore can take an indefinite period of time. The <time limit> parameter of the AWAITIO procedure can be used to ensure that a terminal operator performs an operation within a given period of time. (The terminal must have been opened so as to permit no-wait i/o.)

For example, an application program prompts a terminal operator for an account number. If no entry is made within five minutes, the application program reminds the terminal operator by re-prompting for the account number. To do this, the following is written in the application program:

```
      .
DEFINE five^minutes = 30000D#;
LITERAL timeout = 40;
INT error, .buffer[0:599];
      .
      .
re^prompt:
      .
   buffer ':=' "PLEASE ENTER ACCOUNT NUMBER";
   CALL WRITEREAD ( term^num, buffer, 27, 400, num^read );
   IF < THEN ...
   CALL AWAITIO ( term^num, buffer, num^read, tag, five^minutes );
   IF < THEN ! error occurred
     BEGIN
       CALL FILEINFO ( term^num, error );
       IF error = timeout THEN GOTO re^prompt
       ELSE .....;
     END;
     .
```

The message "PLEASE ENTER ACCOUNT NUMBER" is issued every 5 minutes until the operator responds.

Note that, if the call to AWAITIO had been for any file (i.e., <file number> = -1) and a timeout occurred, the operation pending on the terminal would have to be cancelled before the WRITEREAD could be reinitiated.

MODEMS

Using terminals connected to the system through modems is, for the most part, invisible to the application program. However, the programmer must be aware, when opening a terminal connected though a modem, that the OPEN procedure does not ensure that a communication link has been established.

A CONTROL operation is provided that can be used to signal the application process when a communication link (i.e., incoming call) is established.

BREAK FEATURE

The file system includes special features that permit a terminal operator to signal a process by typing the BREAK key. An example of BREAK usage is when running an application program through the Command Interpreter process; typing BREAK while the application is running returns the Command Interpreter to the command input mode. Because BREAK (if enabled) is constantly monitored by the file system (actually the terminal controller), it is not necessary for the application process to periodically check a terminal for input.

Some characteristics associated with the break feature are:

- BREAK is initially enabled for a process through a SETMODE function (the process that has BREAK enabled is referred to as the "owner" of BREAK).
- BREAK can be enabled for only one process at a time.
- If the terminal is open by the backup process of a NonStop process pair (i.e., via CHECKOPEN by primary or backup open by backup), the backup process will automatically become the owner of BREAK if its primary failed while owning BREAK.
- When BREAK is typed, a system message (-20) is sent to the process (if any) that enabled BREAK. The message is read through the $RECEIVE file and contains the logical device number in binary form of the terminal where BREAK was typed.
- The terminal where BREAK was typed can be set into an access mode (called break mode) so that only operations that have been associated with BREAK (through a call to SETMODE) are allowed.
- Once BREAK is typed, it is disabled, and further breaks on that terminal are ignored. BREAK is automatically re-enabled for the owner when a READ or WRITEREAD procedure is executed to the terminal.
- After typing BREAK, an application not wishing to issue a READ or WRITEREAD to a terminal re-enables BREAK via another SETMODE call.
- Any process using the same terminal as the Command Interpreter, or any other process using BREAK, must perform error recovery for the two errors associated with BREAK: error 111 and error 112.
- If BREAK is typed but not enabled, it is ignored.
- If a process owning BREAK is deleted or fails, BREAK ownership is lost. That is, no process will be informed if the BREAK key is typed.

```
    .
CALL SETMODE ( term^num, set^break^owner, MYPID, normal^mode );
    .
```

MYPID is a Process Control procedure that returns the <cpu,pin> of the caller. Following this call to SETMODE, the file system monitors "term^num" for a break signal. If BREAK is typed, a system BREAK message is sent to this process.

A read is issued to the $RECEIVE file (open as a no-wait file):

```
    .
CALL READ ( recv^fnum, recv^buf, 132 );
    .
```

Then, periodically, $RECEIVE is checked:

```
error := 0;
CALL AWAITIO ( recv^fnum,, num^read,, 0D );
IF = THEN ...                    ! user msg received
ELSE
IF > THEN                        ! system msg received.
  BEGIN
    IF recv^buf = - 20 THEN      ! BREAK message.
      break^received := 1

          flags the fact that BREAK was typed.  The break is
          processed in some other part of the program.

    ELSE
    IF recv^buf = ... THEN       ! some other system message.
  END
ELSE                             ! error
  CALL FILEINFO ( recv^fnum, error );

! if read on $RECEIVE completed, issue another.
IF error <> 40 ! timeout ! THEN
   CALL READ ( recv^fnum, recv^buf, 132 );
```

Note: If a process has BREAK armed on more than one terminal, it should check the logical device number returned in the system BREAK message to identify the source of the break.

Figure 2-20 illustrates the break sequence when a terminal is controlled by a single process.

For example, when an application wanting to use the break feature is to be run through the Command Interpreter program (which also uses BREAK), the application should get the <cpu,pin> and break mode of the current owner when enabling BREAK for itself:

```
.
INT last^owner[0:1],
    last^mode = last^owner [1];
.

.
CALL SETMODE ( home^term^num, set^break^owner, MYPID, normal^mode,
               last^owner );
.
```

An internally defined integer designating the last owner of BREAK is returned in "last^owner", and the mode associated with the last owner is returned in "last^owner[1]" (= "last^mode"). If no process previously had BREAK enabled, zero (0) is returned to "last^owner". BREAK is now enabled for this process (i.e., will receive the BREAK message if break is typed).

Note: The number returned in "last^owner" is NOT the <cpu,pin> of the last owner of BREAK.

When the application no longer wants to receive the BREAK message, it re-enables BREAK for the last owner (the Command Interpreter in this example):

```
.
CALL SETMODE ( home^term^num, set^break^owner, last^owner,
               last^mode );
.
```

At this point, if BREAK is typed, the Command Interpreter will receive the BREAK message.

If each process using BREAK keeps track of the previous owner, BREAK ownership can be passed between any number of processes in an orderly fashion.

Break Mode

By using break mode, a number of processes can access the same terminal, but one process can take exclusive access to that terminal when BREAK is typed.

This is done in three steps:

1. First, when BREAK is enabled, break mode is specified. This means that, after BREAK is typed, the terminal is put in break mode, and only file operations having break access are permitted with the terminal.

If the terminal access mode is "break mode" when the owner of break closes the file and the owner has "break access" specified, the terminal access mode is returned to normal mode. This applies if the close is because of a call to the file system CLOSE procedure or the process control STOP procedure.

Note: Unless more than one process is accessing a terminal, normal access (i.e., <parameter 2> = 0) should be specified.

For example:

```
   LITERAL set^access    = 12,
           break^mode    =  1,
           normal^mode   =  0,
           break^access  =  1,
           normal^access =  0;
     .
```

BREAK is enabled and break mode is specified:

```
     .
   CALL SETMODE ( home^term^num, set^break^owner, MYPID, break^mode,
                  last^owner );
     .
```

Then $RECEIVE is periodically checked for a BREAK message:

```
       .
      CALL READ ( recv^fnum, recv^fnum, 132 );
       .
      error := 0;
      CALL AWAITIO ( recv^fnum,, num^read,, 0D );
      IF = THEN ...                ! user msg received
      ELSE
      IF > THEN                    ! system msg received.
        BEGIN
          IF buffer = - 20 THEN    ! break message.
            BEGIN
```

Break access is specified:

```
              CALL SETMODE ( home^term^num, set^access,,
                             break^access );
                .
                 At this point, any non-break operations to the
                 terminal indicated by "home^term^num" will be
                 rejected.

                 However, this process, because break access was
                 specified, can access the terminal.

                .
                .
```

The SETMODE function to gain exclusive access to a terminal is:

<function> = 12, set terminal access mode and file access type.

<parameter 1> = 1, terminal access mode = break mode.

<parameter 2> = 1, file access mode = break access.

The SETMODE function to relinquish exclusive access to a terminal is:

<function> = 12, set terminal access mode and file access type.

<parameter 1> = 0, terminal access mode = normal mode.

<parameter 2> = 0, file access mode = normal access.

Note: An application program should use this feature only if it has ownership of BREAK. If a process that does not own BREAK is deleted, break mode is not cleared. Other processes accessing the terminal with normal access are then prevented from accessing the terminal.

For example, a process needs temporary exclusive access to a terminal. The following call to SETMODE is made:

```
.
CALL SETMODE ( home^term^num, set^access, break^mode,
               break^access );
.
```

At this point, any other operations flagged as "normal access" to the terminal will be rejected.

When the process no longer requires exclusive access to the terminal, it permits normal access through the following call to SETMODE:

```
.
CALL SETMODE ( home^term^num, set^access, normal^mode,
               normal^access );
.
```

Exclusive access using BREAK is illustrated in figure 2-22.

Error 110 (only break access permitted): This error indicates that BREAK was typed and that break mode was specified when BREAK was enabled (see SETMODE, function 11). The terminal is inaccessible (unless this process uses SETMODE to signal its operations as break access) until the process processing the break calls SETMODE (function 12) to allow normal access to the terminal.

If the process receiving error 110 is not the one that enabled BREAK, then the operation should be retried periodically. If the process has break enabled, then $RECEIVE should be checked for the system BREAK message and appropriate action should be taken.

Note: This error implies that no data was transferred.

Error 111 (operation aborted because of BREAK): This error indicates that BREAK was typed while the current file operation was taking place. The nature of this error indicates that data may have been lost.

If the process receiving error 111 is not the one that enabled BREAK, then the operation should be retried. If a write operation was being performed, then the write can simply be retried. If a read operation was being performed, then a message should be sent, telling the terminal operator to retype the last entry, before retrying the read.

Keep in mind, however, that if more than one process is accessing a terminal and the break feature is used, only break access should be allowed after BREAK is typed. Therefore, subsequent retries are rejected with error 110 until normal access is permitted.

If either of these errors is received by a process not having BREAK enabled, the process should suspend itself for some short period (like ten seconds) before retrying the operation. This can be accomplished by calling the process control DELAY procedure.

If the process has BREAK enabled, then $RECEIVE should be checked for the system BREAK message and appropriate action should be taken.

Preempted by Operator Message (Error 112)

This error can occur only when an application process is using the same terminal as the active operator console device. If the application process is reading from the terminal (using either READ or WRITEREAD) and a message is sent to the operator, the application process's file operation is aborted and the operator message is written. (This is necessary so that operator messages are not inadvertently deferred while some read is occurring on the terminal). Any data entered when the preemption takes place is lost. Therefore, a message should be sent telling the terminal operator to retype the last entry before retrying the read.

SUMMARY OF TERMINAL CONTROL AND SETMODE OPERATIONS

Table 2-7. Terminal CONTROL and SETMODE Operations

Terminal CONTROL Operations

<operation>

1 = forms control:

<parameter> for terminal

0 = form feed (send %014)
> 0 = vertical tab (send %013)

11 = wait for modem connect:

<parameter> = none

12 = disconnect the modem (i.e., hang up):

<parameter> = none

Terminal SETMODE Operations

<function>

6 = set system spacing control:

<parameter 1> = 0, no space
= 1, single space (default setting)

<parameter 2> is not used.

7 = set system auto line feed after receipt of carriage return line termination (default mode is configured):

<parameter 1> = 0, off
= 1, system provides line feed after line termination by carriage return

<parameter 2> is not used.

8 = set system transfer mode (default mode is configured):

<parameter 1> = 0, conversational mode
= 1, page mode

<parameter 2> is not used.

Table 2-7. Terminal CONTROL and SETMODE Operations (cont'd)

```
     <parameter 2> is not used.

14 = set system read termination on interrupt characters (default
     is configured):

     <parameter 1> = 0, no termination on interrupt characters
                        (i.e., transparency mode)
                   = 1, termination on any interrupt character
                        input

     <parameter 2> is not used.

20 = set system echo mode (default is configured).

     <parameter 1> = 0, system does not echo characters as read
                   = 1, system echoes characters as read

     <parameter 2> is not used.

22 = set baud rate:

     <parameter 1> = 0, baud rate = 50
                     1, baud rate = 75
                     2, baud rate = 110
                     3, baud rate = 134.5
                     4, baud rate = 150
                     5, baud rate = 300
                     6, baud rate = 600
                     7, baud rate = 1200
                     8, baud rate = 1800
                     9, baud rate = 2000
                    10, baud rate = 2400
                    11, baud rate = 3600
                    12, baud rate = 4800
                    13, baud rate = 7200
                    14, baud rate = 9600
                    15, baud rate = 19200

     <parameter 2> is not used.

23 = set character size:

     <parameter 1> = 0, character size = 5 bits
                     1, character size = 6 bits
                     2, character size = 7 bits
                     3, character size = 8 bits

     <parameter 2> is not used.
```

The file system provides for data transfers from application processes to line printers in blocks of 0 (blank line) to the maximum number of characters permitted in one line of print.

The following topics are covered in this section:

- General Characteristics of Line Printers
- Summary of Applicable Procedures
- Accessing Line Printers
- Forms Control
- Programming Considerations for the Model 5508 Line Printer
- Programming Considerations for the Model 5520 Line Printer
- Using a Model 5508 or 5520 Printer Over a Phone Line
- Error Recovery
- Summary of Printer CONTROL, CONTROLBUF, and SETMODE Operations

GENERAL CHARACTERISTICS OF LINE PRINTERS

- Line printers are accessed by

  $<device name> or

  $<logical device number>.

- Default file system spacing mode is "post-space" (i.e., space after printing). The spacing mode can be set to "pre-space" (i.e., space before printing) via a SETMODE function.

- A standard VFU tape is supplied with each line printer (see table 2-8 at the end of this section).

- Procedures available for explicitly controlling forms movement are:

  - CONTROL

    Skip to VFU channel or skip a number of lines.

  - CONTROLBUF

    Load programmable VFU (DAVFU) for model 5520 printer.

  - SETMODE and SETMODENOWAIT

    No-space or single-space after printing.

    Disable/enable automatic perforation skip.

- The file system does not provide automatic top-of-form on OPEN or CLOSE; if this is desired, it must be handled by an application process via a call to the CONTROL procedure.

- It is the responsibility of application processes to handle "paper out" and "not ready" conditions.

```
   .
CALL OPEN ( ptr, file^num, excl^acc ); ! exclusive access.
   .
```

Then to print a line of print:

```
   .
CALL WRITE ( file^num, ptr^buffer, 132 );
   .
```

prints 132 characters of "ptr^buffer" on the line printer.

If the printer has a configured line width of 132 characters and the following call is made:

```
   .
CALL WRITE ( file^num, ptr^buffer, 200 );
IF <> THEN ...;
   .
```

an error occurs. The first 132 characters of "ptr^buffer" are printed. On the return from WRITE, the condition code indicator is set to CCL. A subsequent call to FILEINFO would return error 21 (illegal count specified).

If the printer has a configured line width of 132 characters and the following call is made:

```
   .
CALL WRITE ( file^num, ptr^buffer, 40 );
IF <> THEN ...;
   .
```

40 characters of "ptr^buffer" are printed starting at column 1; columns 41 through 132 are left blank.

Note: If the <count written> parameter is present in the call to WRITE, it is returned a count of the number of characters actually printed.

FORMS CONTROL

The file system CONTROL and SETMODE procedures provide the programmer with the capability of controlling forms movement.

The only automatic forms movement provided by the file system is the perforation skip and single space paper movement. Both of these can be disabled through use of the SETMODE procedure. Note that any automatic forms movement always takes place after a line is printed.

The CONTROL procedure is used either to advance forms according to a vertical format tape installed in the printer (or a programmable vertical format, or DAVFU) or to advance forms a specified number of

Overprinting can be accomplished through SETMODE function 6. For example, to overprint a single line of print, the following calls to SETMODE and WRITE are made:

```
LITERAL set^space = 6,
        no^space  = 0,
        space     = 1;

   .
CALL SETMODE ( file^num, set^space, no^space );
   .
```

turns off single spacing.

```
   .
CALL WRITE ( file^num, buffer1, ... );
   .
```

prints the contents of "buffer1". The form does not advance.

```
   .
CALL SETMODE ( file^num, set^space, space );
   .
```

turns on single spacing.

```
   .
CALL WRITE ( file^num, buffer2, ... );
   .
```

prints the contents of "buffer2" over the line just printed (i.e., contents of "buffer1"). The form advances to the next line.

Note: Application programs should use CONTROL and SETMODE to accomplish forms control rather than attempting to embed forms control characters in the print line. The line printer does not recognize the unprintable characters, and therefore an error 218 (interrupt timeout) occurs.

PROGRAMMING CONSIDERATIONS FOR THE MODEL 5508 PRINTER

The subtype for the model 5508 line printer is 3.

Programming Form Length and Vertical Tab Stops

The model 5508 line printer has an electronically programmable form length and vertical tabulation stops.

The number of lines in the form is specified via SETMODE function 25 as an integer within the range of {0:126}. The default for this setting is 66.

The model 5508 printer provides programmable forms length and vertical tab stops, whereas the model 5520 printer provides a 12-channel Direct Access Vertical Format Unit (DAVFU) internal buffer whose contents may be changed by the user program. The model 5508 printer allows the user program to specify forms length and vertical tab stops by means of SETMODE functions; for the model 5520, the user program specifies the contents of the DAVFU (if values other than the defaults are desired) by calling the CONTROLBUF procedure.

User applications which currently run with the model 5508 printer must be modified to run on the model 5520 if they are affected by any of the following differences:

- SETMODE 25 (forms length) is not supported on the model 5520.
- SETMODE 26 (set/clear vertical tab stops) is not supported on the model 5520.
- CONTROL 1 (forms control) is used differently, in some cases, on the two printers.
- The vertical tab (VT) character is not supported on the model 5520.
- Control characters (%00-%37) should not be included in user data sent to the model 5520, since they disable parity error recovery.

In the following discussion, it is assumed that line numbers range from 1 to 254, character locations range from 1 to 218, and VFU channels range from 0 to 11.

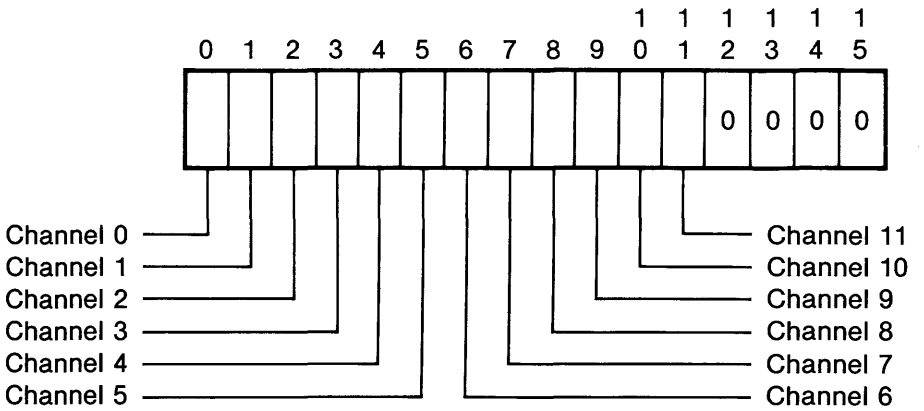
Using DAVFU

The DAVFU specifies forms length and the functions performed by CONTROL operation 1 (forms control). The 5520 default DAVFU is initialized as follows:

VFU channel 0	(top of form/line 1)
VFU channel 1	(bottom of form/line 60)
VFU channel 2	(single space/lines 1-60, top-of-form eject)
VFU channel 3	(next odd-numbered line)
VFU channel 4	(next third line: 1,4,7,10, etc.)
VFU channel 5	(next one-half page)
VFU channel 6	(next one-fourth page)
VFU channel 7	(next one-sixth page)
VFU channel 8	(line 1)
VFU channel 9	(line 1)
VFU channel 10	(line 1)
VFU channel 11	(bottom of paper/line 63)

The default form length is 66 lines with automatic perforation skip mode enabled (first 60 lines printed per page).

The format of each word in the DAVFU buffer is as follows:



For example, %37400 (default for line 31) indicates that channel stops are to be set for channels 2, 3, 4, 5, 6, and 7.

Loading the DAVFU causes the top of form to be reset to the current line.

There must be exactly one channel 0 (top of form) stop defined in the VFU buffer, and at least one stop defined for each of the other VFU channels. File system error 105 (VFU ERROR) is returned on an attempt to load the DAVFU if the VFU buffer is not valid.

The forms length is determined by the number of words in the VFU buffer. The maximum forms length allowed is 254 lines (508 bytes in buffer). If the byte count specified in a DAVFU load exceeds this maximum, or if it is not an even number, file system error 21 (ILLEGAL COUNT) is returned.

Channel 0 is used to indicate which line the printer should skip to if CONTROL operation 1, <parameter> = 0 is issued, or if the operator presses the TOP OF FORM button on the printer. Channel 11 is used to indicate which line the printer should skip to if CONTROL operation 1, <parameter> = 11 is issued, and also to indicate when "paper out" should be reported.

Upon detecting a paper out condition, the printer waits until the line defined as "bottom of paper" has been printed or passed over. This feature makes it possible to complete printing of an entire page before the paper out condition is reported to the application program.

When using the default DAVFU, line 63 is defined as bottom of paper. If the paper is properly aligned in the printer, line 63 is the last line before the perforation and lines 64, 65, and 66 are the first three lines on the new page. Line 1 (top of form) immediately follows as the fourth line on the page.

4. Write a buffer, consisting only of underscore and space characters, to underline the desired parts of the text written in step 2.

This procedure must be followed exactly for the partial line feed to occur. If, for example, the application's second write buffer includes other text on the same line with the underscore and space characters, the line is still overprinted, but the partial line feed is not performed.

The following example illustrates use of the partial line feed for underlining.

```
INT .databuf1[0:11] := ["THIS WILL BE UNDERLINED."],
    .databuf2[0:11] := ["_____________________ "];
       .
       .

! Turn off spacing
CALL SETMODE (filenum, 6, 0);

! Write the text
CALL WRITE (filenum, databuf1, 24);

! Turn spacing back on
CALL SETMODE (filenum, 6, 1);

! Underline the text
CALL WRITE (filenum, databuf2, 24);
```

Condensed and Expanded Print

The model 5520 printer provides condensed and expanded print capabilities in addition to the standard spacing of 10 characters per inch.

The condensed print option allows the 5520 to print with a horizontal pitch of 16.7 characters per inch. Condensed print may be selected by calling SETMODE function 68, <parameter 1> = 1.

The expanded print option (double width) provides a horizontal pitch of 5 characters per inch. The user application program may select expanded print by calling SETMODE function 68, <parameter 1> = 2.

Normal printing may be reenabled by calling SETMODE function 68, <parameter 1> = 0.

For example:

```
! Select condensed print
CALL SETMODE (filenum, 68, 1);
```

Error 105 (VFU ERROR) indicates that the VFU buffer is invalid. This can occur for the following reasons: 1) more than one stop was defined for channel 0 (top of form); 2) no stops were defined for one or more channels; 3) bits 12 through 15 of each word were not zeros.

Error 120 (DATA PARITY ERROR) indicates a non-recoverable data parity error. For details, see "Data Parity Error Recovery" following this section.

Error 121 (DATA OVERRUN) means that the buffer overflowed while data was being sent to the printer. This indicates a hardware or microcode problem.

Error 190 (DEVICE ERROR) indicates one of the following conditions: 1) invalid status returned from printer; 2) "buffer full" status lasted longer than 10 seconds; 3) no shuttle motion; 4) character generator absent; 5) VFU fault which is not recoverable; 6) VFU channel error.

Error 191 (DEVICE POWER ON) indicates that the printer powered on while the file was open. For details, see "DEVICE POWER ON Error" following the next section.

Data Parity Error Recovery

Automatic parity error recovery is supported for the 5520 printer. If a parity error is detected, the i/o software will attempt to recover unless one of the following conditions exists:

- A parity error persisted after the retry count was exhausted.
- A parity error occurred while the device was in an offline state (not ready or paper out).
- A parity error occurred on a request for status immediately following a write of data (a parity error may also have occurred in the data).
- A parity error occurred during a write of data that contained embedded control characters.

Control characters (%00-%37) should not be included in data sent to the 5520 printer, since they disable parity error recovery. The i/o software provides the appropriate line termination and all escape sequences. The 5520 recognizes the following control characters: line feed (%12), form feed (%14), carriage return (%15), and escape (%33). Any other control characters are printed as a space. Escape is used as the first character in all escape sequences. Any unrecognized escape sequence sent to the printer is assumed to be a five-character sequence, and results in the printing of a non-standard character (%206), followed by paper movement equivalent to one line feed.

```
CALL CONTROL (filenum, 11);    ! answer the phone
        .
        .
CALL CONTROL (filenum, 12);    ! hang up the phone
```

AUTOANSWER or CTRLANSWER mode can be specified as configuration parameters in SYSGEN (see the NonStop System Management Manual or the NonStop II System Management Manual) or by using SETMODE function 29:

```
CALL SETMODE (filenum, 29, 0); ! CTRLANSWER mode
        .
        .
CALL SETMODE (filenum, 29, 1); ! AUTOANSWER mode
```

Unlike other SETMODE functions, this one remains in effect even after the file is closed.

The following configuration parameters are needed in order to use a modem with the model 5508 printer:

LP5508M	MODEM	EIA	BAUD300 or BAUD1200	AUTOANSWER or CTRLANSWER

The following configuration parameters are needed in order to use a modem with the model 5520 printer:

LP5520M	MODEM	EIA	BAUD300 or BAUD1200	AUTOANSWER or CTRLANSWER

Further information on system configuration for these devices can be found in the SYSGEN section of the NonStop System Management Manual or the NonStop II System Management Manual.

ERROR RECOVERY

The following errors require special consideration for line printers:

- 100 not ready
- 200-255 path errors

Additionally, consideration is necessary when using no-wait i/o and permitting more than one concurrent i/o operation. It is possible, when initiating a number of operations, that some can fail while subsequent operations do not. In that case, lines may be missing and, if reprinted, would be out of order.

SUMMARY OF PRINTER CONTROL, CONTROLBUF, AND SETMODE OPERATIONS

Table 2-8. Line Printer CONTROL, CONTROLBUF, and SETMODE Operations

Line Printer CONTROL Operations

1 = forms control:

<parameter> for printer (subtype 0, 2, or 3)

0	= skip to VFU channel 0	(top of form)
1 - 15	= skip to VFU channel 1	(single space)
16 - 79	= skip <parameter> - 16 lines	

<parameter> for printer (subtype 1 or 5)

0	= skip to VFU channel 0	(top of form)
1	= skip to VFU channel 1	(bottom of form)
2	= skip to VFU channel 2	(single space, top-of-form eject)
3	= skip to VFU channel 3	(next odd-numbered line)
4	= skip to VFU channel 4	(next third line: 1, 4, 7, 10, etc.)
5	= skip to VFU channel 5	(next one-half page)
6	= skip to VFU channel 6	(next one-fourth page)
7	= skip to VFU channel 7	(next one-sixth page)
8	= skip to VFU channel 8	(user-defined)
9	= skip to VFU channel 9	(user-defined)
10	= skip to VFU channel 10	(user-defined)
11	= skip to VFU channel 11	(user-defined)
16 - 31	= skip <parameter> - 16 lines	

<parameter> for printer (subtype 4) (default DAVFU)

0	= skip to VFU channel 0	(top of form/line 1)
1	= skip to VFU channel 1	(bottom of form/line 60)
2	= skip to VFU channel 2	(single space/lines 1-60, top-of-form eject)
3	= skip to VFU channel 3	(next odd-numbered line)
4	= skip to VFU channel 4	(next third line: 1, 4, 7, 10, etc.)
5	= skip to VFU channel 5	(next one-half page)
6	= skip to VFU channel 6	(next one-fourth page)
7	= skip to VFU channel 7	(next one-sixth page)
8	= skip to VFU channel 8	(line 1)
9	= skip to VFU channel 9	(line 1)
10	= skip to VFU channel 10	(line 1)
11	= skip to VFU channel 11	(bottom of paper/line 63)
16 - 31	= skip <parameter> - 16 lines	

Table 2-8. Line Printer CONTROL, CONTROLBUF, and SETMODE Operations (cont'd)

Line Printer SETMODE Operations (cont'd)

<function>

22 = line printer (subtype 3 or 4), set baud rate:

<parameter 1> = 0, baud rate = 50
1, baud rate = 75
2, baud rate = 110
3, baud rate = 134.5
4, baud rate = 150
5, baud rate = 300
6, baud rate = 600
7, baud rate = 1200
8, baud rate = 1800
9, baud rate = 2000
10, baud rate = 2400
11, baud rate = 3600
12, baud rate = 4800
13, baud rate = 7200
14, baud rate = 9600
15, baud rate = 19200

<parameter 2> is not used.

25 = line printer (subtype 3), set form length:

<parameter 1> = length of form in lines

<parameter 2> is not used.

26 = line printer (subtype 3), set/clear vertical tabs:

<parameter 1> >= 0, line where tab is to be set
= -1, clear all tabs (except line 0)

Note: A vertical tab stop always exists at line 0 (top of form).

<parameter 2> is not used.

27 = set system spacing mode:

<parameter 1> = 0, post-space (default setting)
= 1, pre-space

<parameter 2> is not used.

Table 2-8. Line Printer CONTROL, CONTROLBUF, and SETMODE Operations (cont'd)

Line Printer SETMODE Operations (cont'd)

37 = line printer (subtype 1, 4, or 5), get device status (cont'd):

<last params>[0] for printer (subtype 1 or 5) (cont'd)

.<13> = DPE, device parity error	0 = parity OK 1 = parity error
.<14> = NOL, not on line	0 = on line 1 = not on line
.<15> = NRY, Not ready	0 = ready 1 = not ready

All other bits are undefined.

Note that Ownership, Interrupt Pending, Controller Busy, and Channel Parity errors are not returned in <last params>; your application program "sees" them as normal file errors. Also note that CID must be checked when PMO, BOF, and TOF are tested, since the old cable version does not return any of these states.

<last params> for printer (subtype 4)

<last params>[0] = primary status returned from printer:

.<9:11> = full status field	0 = partial status 1 = full status 2 = full status / VFU fault 3 = reserved for future use 4 = full status / data parity error 5 = full status / buffer overflow 6 = full status / bail open 7 = full status / auxiliary status available
.<12> = buffer full	0 = not full 1 = full
.<13> = paper out	0 = OK 1 = paper out
.<14> = device power on	0 = OK 1 = POWER ON error

The file system provides for data transfers between magnetic tape files and application processes in records of 24 to 4096 bytes.

Topics covered in this section are:

- General Characteristics of Magnetic Tape Files
- Summary of Applicable Procedures
- Accessing Tape Units
- Tape Concepts
  - BOT and EOT Markers
  - Files
  - Records
- 5106 Tri-Density Tape Subsystem
- Error Recovery
- Summary of Magnetic Tape CONTROL Operations
- Seven-Track Magnetic Tape Conversion Modes

GENERAL CHARACTERISTICS OF MAGNETIC TAPE FILES

- Individual files on a magnetic tape are not accessed explicitly; instead, the magnetic tape unit itself is accessed by:

  $<device name> or

  $<logical device number>.

- Procedures are provided that permit the application to write, locate, and read any number of files desired.

- It is the responsibility of the application program to delimit a file on tape by explicitly writing an end-of-file mark (i.e., closing a magnetic tape file following a write to tape does NOT write an end-of-file mark).

- The CONTROL and CLOSE procedures provide four options for rewinding tape. These options are fully described in Tables 2-1 and 2-9.

- To ensure the integrity of the data written on tape, the file system pads write operations of less than 24 bytes with "null" (0) characters. The number of pad bytes is 24 minus <write count>, so that the minimum physical record ever written on tape is 24 bytes (e.g., a <write count> of 0 causes a record containing 24 null bytes to be written on tape).

- The file system permits reads and WRITEUPDATEs of as few as two bytes (this permits tapes written on non-Tandem systems to be read or edited). WRITEUPDATEs are not allowed on the Tri-Density Tape Drive. To ensure the integrity of data read from tape, however, the minimum read operation should be for at least 24 bytes.

- Multi-reel files, if desired, must be implemented by the application program.

SUMMARY OF APPLICABLE PROCEDURES

The following procedures are used to perform input/output operations with magnetic tapes:

DEVICEINFO	provides the device type and configured record length for a designated magnetic tape unit
OPEN	establishes communication with a file
READ	reads information from an open file
WRITE	writes information to an open file
WRITEUPDATE	is used to replace an existing record (not supported on 5106 Tri-Density Tape Drive)
CONTROL	executes the following operations to a magnetic tape: • write end-of-file mark • rewind (load/unload,online/offline,wait/don't wait) • record spacing (forward and backward) • file spacing (forward and backward)
AWAITIO	waits for completion of an outstanding i/o operation pending on an open file
CANCELREQ	cancels the oldest outstanding operation, optionally identified by a tag, on an open file
FILEINFO	provides error information and characteristics about an open file
SETMODE	sets/clears the translation technique option (7-track tape only) and the short write treatment option. Selects tape density for 5106 Tape Drive.
SETMODENOWAIT	is used the same as SETMODE except in a no-wait manner on an open file
CLOSE	stops access to an open file and, optionally, rewinds the tape

ACCESSING TAPE UNITS

Like any other file, a magnetic tape unit is accessed through the OPEN procedure. For example, to access a magnetic tape unit that is assigned the device name "$TAPE1", the following could be written in an application program:

BOT and EOT Markers

The BOT (beginning of tape) and EOT (end of tape) markers delimit the useful area on tape. When a tape is loaded and initially made ready, the tape is positioned with the read/write heads located slightly past the BOT marker.

If a backspace files (CONTROL operation 8) or a backspace records (CONTROL operation 10) is being executed and the BOT marker is encountered, tape motion stops and a beginning-of-tape indication (FILEINFO error 154) is returned to the application process.

Crossing the EOT marker in either direction never stops tape motion and never terminates an i/o operation. However, once the EOT marker is passed when writing in the forward direction, the application receives an indication (CCL, FILEINFO error 150) at the completion of each write operation. This indication is returned with each write operation until the EOT marker is passed in the reverse direction.

Note: Because the relationship of the read/write head to the transducer that detects the EOT marker varies from tape unit to tape unit, the EOT indication is not returned when reading. A convention (such as writing two consecutive EOF marks) should be established for the computer site to designate the physical end of tape.

Files

By convention, a file on magnetic tape consists of a number of records followed by an end-of-file mark. It is the responsibility of the application program to explicitly write an EOF mark on tape (using CONTROL, operation 2) to terminate a file.

It is also the responsibility of the application to provide a means of detecting the last file on tape. Typically, this is done by writing two consecutive EOF marks. Naturally, any other programs reading the tape must be aware of such a convention.

The file system provides (as a parameter to the CONTROL procedure) the ability to space forward and backward a specified number of files (i.e., EOF marks).

There are two considerations when spacing files:

- Forward space files stops only after the specified number of EOF marks have been encountered.
- Backward space files stops only after the specified number of EOF marks have been encountered or the BOT marker is detected.

Data is written to tape using the file system WRITE or WRITEUPDATE procedure. The WRITE procedure is typically used when sequentially appending information on the tape. The WRITEUPDATE procedure is used when changing an individual record on tape.

WRITEUPDATE is not allowed on the 5106 Tri-Density Tape Drive.

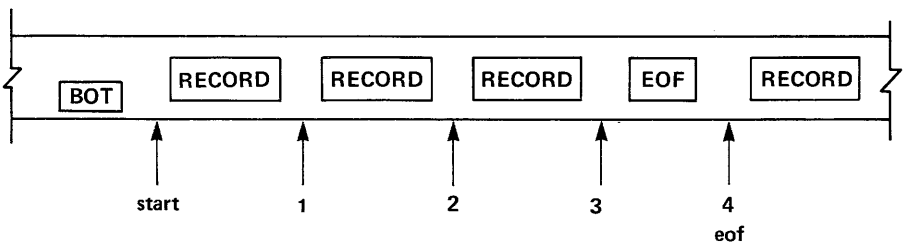
It is important to note that the new record written by the WRITEUPDATE procedure must be exactly the same size as the record being replaced; otherwise, a subsequent error will occur. Also, there is a practical limit of five as to the number of times WRITEUPDATE should be performed on the same record.

Data is read from tape using the file system READ procedure. Any time a read is executed from the tape (even if 0 bytes is specified), the tape spaces one full record. Any one read from a tape is limited to one record on tape.

As an example of tape movement when reading: a file on tape consists of three records, and each record contains 1024 bytes. Repeated reads of 2048 bytes are executed, as follows:

```
    .
LITERAL eof = 1;
    .
    .
loop:
    .
CALL READ ( tape^num, buffer, 2048, num^read );
IF = THEN GOTO loop
ELSE
  BEGIN
    CALL FILEINFO ( tape^num, error );
    IF error = eof THEN .... ! end-of-file encountered.
    ELSE  ....; ! trouble.
  END;
```

Reads one through three each transfer 1024 bytes into "buffer", return 1024 in "num^read", and set the condition code to CCE. Read four encounters an EOF mark. Nothing is transferred into "buffer", 0 is returned to "num^read", and the condition code is set to CCG.



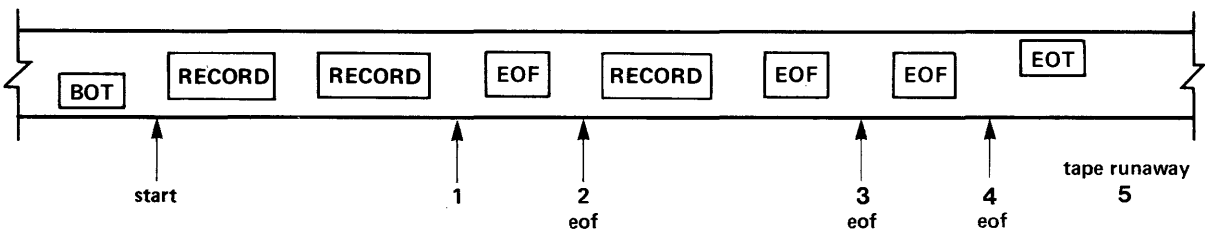
There are a number of considerations when spacing records:

- Forward space records always stops when an EOF mark is read (the tape is positioned with the read/write head past the EOF mark). An indication (CCG, FILEINFO error 1) is returned to the application program.
- Backward space records always stops when an EOF mark is read (the tape is positioned with the read/write head preceding the EOF mark). An indication (CCG, FILEINFO error 1) is returned to the application program.
- Backward space records always stops when the BOT marker is detected (the tape is positioned with the read head preceding the first record on tape). An indication (CCL, FILEINFO error 154) is returned to the application program.

The following examples show how the tape is positioned in relation to the read/write heads following various record spacing operations.

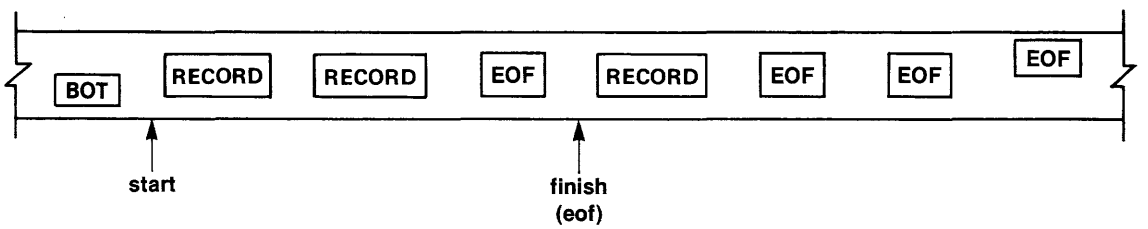
Example 1. Repeated space forwards (CONTROL operation 9) of two records are performed (tape is positioned at BOT):

```
loop:  CALL CONTROL ( tape^file, 9, 2 );
       GOTO loop;
```



Example 2. A space forward of 10 records is performed (tape is positioned at BOT):

```
CALL CONTROL ( tape^file, 9, 10 );
```



The operation stops when an EOF mark is read.

PROGRAMMING CONSIDERATIONS FOR THE 5106 TRI-DENSITY TAPE SUBSYSTEM

The subtype for the 5106 Tri-Density Tape Subsystem is 2. The 5106 Tape Subsystem provides 800 bpi NRZI, 1600 bpi PE, and 6250 bpi GCR recording modes. The correct density is automatically determined during read operations, and allows recording density to be set either programmatically or through hardware switches on its operator panel.

Downloading the Microcode

The Model 3206 Tape Controller (required for the 5106 Tape Subsystem) is downloadable. Thus, the microcode that drives the controller can be loaded from a file on the system disc to the controller over the input/output channel. Whenever the processor which contains the primary i/o process is loaded, or after a controller power failure, the GUARDIAN operating system automatically downloads microcode from the system disc. Then the controller executes commands using the downloaded microcode. An explicit request to download the controller microcode can be made using the PUP LOADMICROCODE command. Refer to the NonStop or NonStop II System Operations Manual.

Download Operation

Associated with the controller is a disc subvolume named $SYSTEM.M<hpn> that contains all possible microcode files necessary for all hardware and prom versions of the controller board to be loaded. This subvolume is placed on the system disc by the INSTALL program. When downloading is required, the operating system searches this subvolume for the appropriate file. The names of the individual files in this subvolume are denoted in this format:

$SYSTEM.M<hpn>.<hhhppp>

where

hpn	is the highest assembly part number of the controller board to be loaded.
hhh	is the hardware revision number of the board.
ppp	is the prom revision number.

When the processor which contains the primary i/o process is loaded or when following a controller power failure, or when a PUP LOADMICROCODE is issued, the operating system attempts to download the microcode as follows:

1. The operating system interrogates the controller to obtain the highest assembly part number for the appropriate controller board, and the proper hardware and prom revision numbers.

Controller Downloading Errors

If a controller error occurs during downloading, a MICROCODE LOADING FAILURE message appears on the operator console. An error indicating the type of failure will be returned to the application program.

At this point, the controller will be executing its resident controller microcode. It cannot perform write, status reporting, or density-setting operations, or several spacing functions. It is still possible, however, to restore files from the 5106 Tape Drive using the existing resident controller microcode and operating system simulation of certain spacing commands.

Selecting Tape Density

On the Model 5106 Tape Drive, READ density is determined automatically by the tape drive formatter when it reads the "id burst" at the beginning of the tape. The default WRITE density is read from the density switches on the operator panel. Programs, however, may override this default WRITE density by calling the SETMODE procedure, with a <function> value of of 66 and <parameter 1> set to indicate the density as follows:

<parameter 1>	Density (bpi)
0	800 (NRZI)
1	1600 (PE)
2	6250 (GCR)
3	As indicated by switches on the tape drive

Immediately after a tape is first opened, the tape driver sets the WRITE density to switch control. A subsequent call to SETMODE 66, before the tape has moved, causes the driver to pass the new density selection to the controller. The new density takes effect with the first write operation after the tape is positioned at the BOT (Beginning of Tape) marker.

Following a controller power failure, the density selection is passed to the controller when the controller microcode is reloaded.

Controller Self-Test Failure

While the 3206 controller is otherwise idle, it periodically initiates several tests against its own hardware. If any test determines that a fatal controller error has occurred, the controller enters hard-failure mode. When this mode is in effect, the system rejects all user requests except the PUP STATUS CONTROLLER command, transmits the FATAL CONTROLLER ERROR message to the operator console, and returns File System Error 224 (controller error) to the application program. To clear the controller's error counters and cause it to return to normal operation, a PUP LOADMICROCODE command, a controller power-on, or a processor reset is necessary.

Error 153 (Drive Power On)

When power is restored after a drive power failure, the operating system automatically places the tape drive online again (with the tape at the BOT marker for the 5106 Tape Subsystem) and returns File System Error 153 to the application program. For other tape drives, the tape is left positioned where it stopped. Now, the application program must either restart the entire tape or reposition to the proper file and record (using counters that it has maintained).

If the application program receives error 100, 212, or 218 immediately after recovery from a drive power failure error, either the tape drive has again lost power or the tape was removed from the drive during the power failure. The error indicates the point at which power was lost.

Error 193 (Invalid or Missing Microcode Files)

This error applies only to the 5106 Tape Subsystem. If the operating system cannot locate either of the controller microcode files, cannot read either of these files because of disc file errors, or cannot download from them because they are not formatted properly, the MICROCODE LOADING FAILURE message appears at the console and the application program receives File System Error 193. (The message appears once for the primary file and once for the backup.) For information about corrective action, see "Invalid or Missing Microcode Files" earlier in this section.

Error 212 (EIO Instruction Failure)

A controller failure failure has occurred (path error). The file operation stopped at some indeterminate point, and the tape may have moved. This error may also indicate a possible controller power failure. (See File System Error 153.)

Error 218 (Interrupt Timeout)

A controller failure or channel failure has occurred. This error may also indicate a possible controller power failure (See File System Error 153).

Error 224 (Controller Error)

This error applies only to the 5106 Tape Subsystem. Certain errors cause the controller to respond with the same error indication until the controller is reset by a PUP LOADMICROCODE command, processor reset, or power failure, returning File System Error 224 (controller error) to the application program. See "Controller Self-Test" below.

SUMMARY OF MAGNETIC TAPE CONTROL OPERATIONS

Table 2-9. Magnetic Tape CONTROL Operations

<operation>

2 = write end-of-file:

<parameter> = none

3 = rewind and unload, don't wait for completion:

<parameter> = none

4 = rewind, take offline, don't wait for completion:
(This option is not available on the 5106 Tape Drive)

<parameter> = none

5 = rewind, leave online, don't wait for completion:

<parameter> = none

6 = rewind, leave online, wait for completion:

<parameter> = none

7 = space forward files:

<parameter> = number of files {0:255}

8 = space backward files:

<parameter> = number of files {0:255}

9 = space forward records:

<parameter> = number of records {0:255}

10 = space backward records:

<parameter> = number of records {0:255}

Table 2-10. ASCII Equivalents to BCD Character Set

BCD TAPE (OCTAL)	BCD MEMORY (OCTAL)	CHARACTER	ASCII (OCTAL)
0	Not Used	Not Used	Not Used
1	1	1	61
2	2	2	62
3	3	3	63
4	4	4	64
5	5	5	65
6	6	6	66
7	7	7	67
10	10	8	70
11	11	9	71
12	0	0	60
13	13	#	43
14	14	@	100
15	15	'(apostrophe)	47
16	16	=	75
17	17	"	42
20	60	space	40
21	61	/	57
22	62	S	123
23	63	T	124
24	64	U	125
25	65	V	126
26	66	W	127
27	67	X	130
30	70	Y	131
31	71	Z	132
32	72	\	134
33	73	,(comma)	54
34	74	%	45
35	75	_(underscore)	137
36	76	>	76
37	77	?	77
40	40	-(minus)	55
41	41	J	112
42	42	K	113
43	43	L	114
44	44	M	115
45	45	N	116
46	46	O	117
47	47	P	120
50	50	Q	121
51	51	R	122

BINARY3TO4 converts each block of three 8-bit memory bytes to four 6-bit tape characters.

The following example illustrates the use of this conversion mode.

A0:A7, B0:B7, and C0:C7 represent three 8-bit memory bytes. These three bytes become four 6-bit tape characters when writing to tape.

```
A0 A1 A2 A3 A4 A5 A6 A7
B0 B1 B2 B3 B4 B5 B6 B7
C0 C1 C2 C3 C4 C5 C6 C7
```

becomes

```
A0 A1 A2 A3 A4 A5
A6 A7 B0 B1 B2 B3
B4 B5 B6 B7 C0 C1
C2 C3 C4 C5 C6 C7
```

When reading from tape, four 6-bit tape characters become three 8-bit bytes.

```
A0 A1 A2 A3 A4 A5
B0 B1 B2 B3 B4 B5
C0 C1 C2 C3 C4 C5
D0 D1 D2 D3 D4 D5
```

becomes

```
A0 A1 A2 A3 A4 A5 B0 B1
B2 B3 B4 B5 C0 C1 C2 C3
C4 C5 D0 D1 D2 D3 D4 D5
```

The maximum record size using the BINARY3TO4 conversion mode is 3072 bytes; parity is odd. Use the number of 8-bit memory bytes to specify a byte count in a read or write.

BINARY1TO1 converts each 8-bit memory byte to one 6-bit tape character. This mode causes the first two bits of every memory byte to be lost when writing to tape.

For example, when writing to tape, the memory byte

A0 A1 A2 A3 A4 A5 A6 A7

becomes

A2 A3 A4 A5 A6 A7

When reading from tape, the first two bits are always zero. For example,

A0 A1 A2 A3 A4 A5

becomes

00 00 A0 A1 A2 A3 A4 A5

The maximum record size in the BINARY1TO1 conversion mode is 4096 bytes; parity is odd.

Selecting the Conversion Mode

Function 33 of the SETMODE procedure selects the conversion mode for 7-track only. The values that specify the conversion modes are:

<parameter 1> = 0, ASCIIBCD
1, BINARY3TO4
2, BINARY2TO3
3, BINARY1TO1

<parameter 2> is not used.

See section 2.3 for discussion of the SETMODE procedure.

Information on system configuration and selection of the default conversion mode for the seven-track tape drive is available in the NonStop System Management Manual or the NonStop II System Management Manual.

The file system provides for transfers from card readers to application processes in blocks of 0 (skip card) to the maximum number of characters required to read a card.

The following topics are covered in this section:

- General Characteristics of Card Readers
- Summary of Applicable Procedures
- Card Reader Access
- Error Recovery

GENERAL CHARACTERISTICS OF CARD READERS

- Card readers are accessed by

  $<device name> or

  $<logical device number>.

- There are three read modes (ASCII, column binary, and packed binary).

- End-of-file indication is available in ASCII read mode. It is "EOF!" in columns 1-4, followed by 76 blank columns.

- End-of-file is not defined for other read modes.

- Application processes must handle the "not ready" condition.

- All Tandem-supplied programs, when accessing a card reader, open it with exclusive access (OPEN, <flags>.<9:11> = 1).

- The card reader device type is 8.

SUMMARY OF APPLICABLE PROCEDURES

The following procedures are used when performing input operations with a card reader:

DEVICEINFO	returns the device type and record length
OPEN	establishes communication with a file
SETMODE	is used to set the card reader read mode
SETMODENOWAIT	is used the same as SETMODE except in a no-wait manner on an open file
READ	is used to read a card

- Column-Binary

  This mode is set by SETMODE function 21, <parameter 1> = 1. In the column-binary mode, each column image is returned right-justified in one word (i.e., two adjacent bytes). Word.<0:3> is 0; the top row of the card is returned in word.<4>.

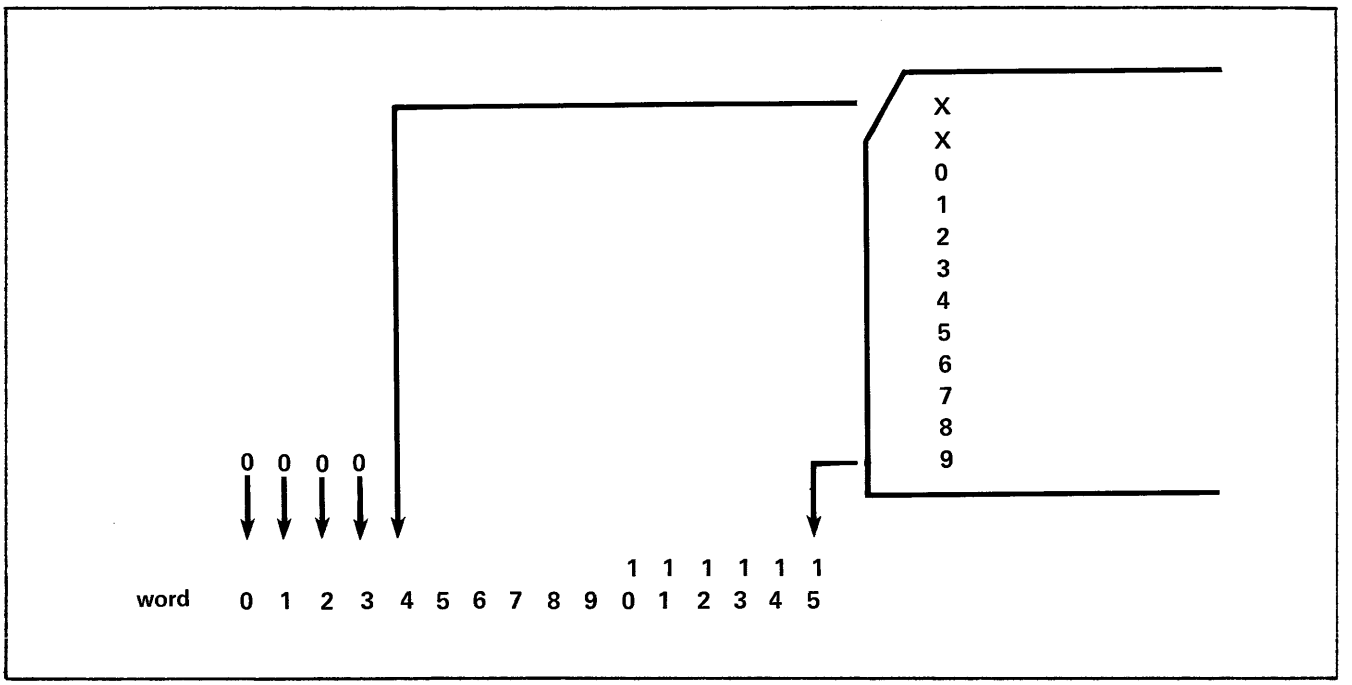


Figure 2-23. Column-Binary Read Mode for Cards

In the column-binary mode, a <read count> of 160 is required to fully read an 80-column card.

```
LITERAL excl^acc = %20;


CALL OPEN ( card^fname, card^fnum, excl^acc );
IF <> THEN ... ; ! error.
   .
```

Then to read a card, the READ procedure is called:

```
   .
CALL READ ( card^fnum, card^buffer, 80 );
IF > THEN ... ; ! end-of-file.
IF < THEN ... ; ! error.
   .
```

returns 80 bytes of ascii data to "card^buffer".

To change the read mode to packed-binary, the following call to SETMODE is made:

```
LITERAL set^read^mode = 21,
        packed^binary =  1;

CALL SETMODE ( card^fnum, set^read^mode, packed^binary );
IF < THEN ... ; ! error.
```

further reads of $CARDRDR will return the card image in packed-binary format.

To close the card reader, the CLOSE procedure is called:

```
   .
CALL CLOSE ( card^fnum );
   .
```

ERROR RECOVERY

The following errors require special consideration:

- 100 not ready
- 145 motion check
- 146 read check error
- 147 invalid Hollerith
- 200-255 path errors

Not Ready

The "not ready" error indicates

- Power Off or
- Hopper Empty

Read Check

This indicates that the card reader hardware has signalled a read check. A possible cause of this condition is a card read hardware malfunction.

The recovery procedure for this error is to stop reading cards, instruct the operator to take the last card through the read station and place it in the input hopper so that it will be the next card read, then resume reading.

If the error persists, then consider the error to be fatal.

Invalid Hollerith

This error can occur in ASCII read mode only. It indicates that a column was read that did not contain a valid Hollerith card code. Specifically, rows one through seven (1-7) contain more than one punch. If the read was for less than a full card, only the first <read count> columns are checked for valid Hollerith codes.

There is no recovery procedure for this error except to stop reading cards, then instruct the operator that the last card through the read station has an invalid Hollerith code.

Path Errors

If a path error is detected and is either error 200 or 201, the operation never got started (card did not feed). These errors can simply be retried.

If a path error is detected and is one of errors 210-231, the operation failed at some indeterminate point. Therefore, a card may have been fed. The simplest way to recover from these errors is to restart the card read operation from the beginning.

The file system provides for data transfers between application processes in blocks of 0 to slightly more than 32,000 characters. Interprocess communication is accomplished via standard file system procedure calls.

The programming described in this section requires first that the processes be created. For the definition of a process and a description of how processes are created and controlled, see section 3, "Process Control".

The following topics are covered in this section:

- General Characteristics of Interprocess Communication
- Summary of Applicable Procedures
- Communication
- $RECEIVE File
  - No-Wait I/O
  - OPEN, CLOSE, CONTROL, SETMODE, RESETSYNC, and CONTROLBUF System Messages
  - Communication Type
- Process Files
- Sync ID
- Interprocess Communication Example
- System Messages
- Error Recovery

GENERAL CHARACTERISTICS OF INTERPROCESS COMMUNICATION

- A file is opened to receive and, optionally, reply to messages from all other processes, using

  $RECEIVE.

  The device type for $RECEIVE is 2.

- A file is opened to send messages to a process and, optionally, wait for a reply, using a "process ID". If the open is to a process or a process pair whose name is in the Process-Pair Directory (PPD), the process ID consists of a symbolic

  $<process name> or \ <sys#> <process name>.

  If a network ID is used, <sys#> is the system number.

  The process name form of the process ID can be further qualified at file open time by the addition of one or two optional qualifier names. This provides for process file names of the form:

- The file number of the sender's file that sent the message. The file number parameter allows the receiver to identify separate opens by the same sender. The value returned in <file number> is the same as the file number used by the sender to send the message. This parameter is returned by RECEIVEINFO.

- The number of reply bytes expected by the sender (i.e., read count value). The <read count> parameter allows the receiver process to identify the type of request being made by the sender. If <read count> = 0, a WRITE request or WRITEREAD request with a read count of zero was made; if <read count> > 0, then the requestor performed a WRITEREAD request of <read count> bytes. This information can be used to determine if the sender is simply sending data (e.g., if <read count> = 0, then sender is "listing") or expects a reply (e.g., if <read count> > 0, then sender is "prompting"). This parameter is returned by RECEIVEINFO.

- Messages from the Command Interpreter (such as the startup parameter message) are read via the $RECEIVE file.

- System messages are read through the $RECEIVE file. The receipt of a system message causes a condition code of CCG to be returned when the read on $RECEIVE completes. Note that messages from the Command Interpreter are not system messages, and therefore do not cause a CCG indication.

- A process specifies at file open time whether or not it wishes to receive OPEN, CLOSE, CONTROL, SETMODE, RESETSYNC, and CONTROLBUF system messages:

  - The OPEN and CLOSE system messages are received by a process when it is opened or closed.

  - The CONTROL, SETMODE or SETMODENOWAIT, and CONTROLBUF procedures can be called for files representing processes. The process referenced by the call is sent a system message containing the CONTROL, SETMODE, or CONTROLBUF parameters.

  - For an explanation of the RESETSYNC procedure, see section 5.

  If a process elects to receive these messages, the process ID of the application process that called OPEN, CLOSE, CONTROL, SETMODE, SETMODENOWAIT, RESETSYNC, or CONTROLBUF is obtained by calling the LASTRECEIVE or RECEIVEINFO procedure.

SETMODENOWAIT
: is used the same as SETMODE, except in a no-wait manner on an open file

CLOSE
: stops access to an open file

COMMUNICATION

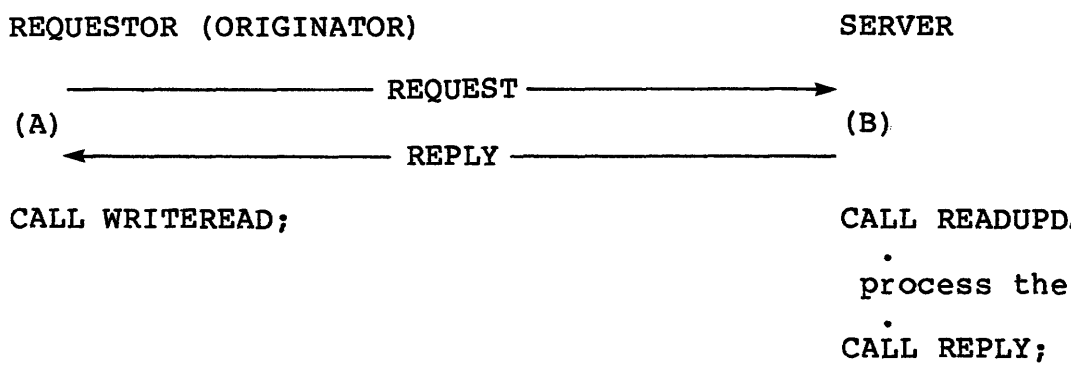
There are two types of communication possible between processes:

- One-way communication,

where the destination reads a message in a call to the READ procedure. The originator's WRITE completes when the destination's READ completes.

Note: If the originator sends a message in a call to the WRITEREAD procedure, the WRITEREAD completes when the destination's READ completes. No data is returned to the originator.

- Two-way communication,



where the "server" process picks up a message in a call to the READUPDATE procedure then, subsequently, replies to the message in a call to the REPLY procedure. The "requestor" sends the message and waits for the reply by calling the WRITEREAD procedure. The WRITEREAD completes when the server's REPLY completes.

Note: If the requestor sends a message in a call to the WRITE procedure, the WRITE completes when the destination's REPLY completes. No data is returned to the requestor.

It is also possible for the server to queue requests before replying:

$RECEIVE FILE

The $RECEIVE file is used by a process to read and optionally reply to messages from other processes and to read messages from the operating system.

Like any other file, the $RECEIVE file must be opened to be accessed. Unlike other input (ie, read) files, however, reading $RECEIVE does not solicit information from some input device. Instead, unsolicited messages are sent to a process via its process ID or process name.

The first message (or series of messages) that a process that was created by a Command Interpreter should expect is the parameter message. This message, depending on the particular application, may contain various parameters to be used by the application process. The first word of the message contains a value of -1. See section 11, "Command Interpreter/Application Interface", for the message format.

The following should be taken into consideration when opening the $RECEIVE file:

- Is no-wait i/o desired?
- Are OPEN, CLOSE, CONTROL, SETMODE, RESETSYNC, and CONTROLBUF system messages desired?
- Is two-way communication to be performed and, if so, is the opener going to perform message queueing and what is the maximum number to be queued?

No-Wait I/O

If only the parameter message is to be read or if it is permissible to have the process suspended while waiting for an incoming message, then the $RECEIVE file should be opened with wait i/o specified (the default).

However, if the process must execute concurrently with the receipt of messages, the $RECEIVE file must be opened with no-wait i/o specified. If no-wait i/o is specified, a read is issued to the $RECEIVE file, and the AWAITIO procedure is called periodically to check for an incoming message. This technique is quite useful for two reasons:

- Process execution continues with a minimum amount of time wasted waiting for messages that may not be present.
- If AWAITIO is called for any file (i.e., <file number> = -1), then an incoming message can be received while waiting for some other no-wait i/o operation to complete.

For example:

Communication Type

Whether or not a process is to perform two-way communication via the $RECEIVE file is indicated by the state of the <receive depth> parameter of the OPEN procedure.

If <receive depth> = 0, one-way communication is indicated. The receiver can only accept incoming messages; replies cannot be issued. Incoming messages must be read via calls to the READ procedure. Calls to READUPDATE and REPLY are not permitted.

If <receive depth> >= 1, two-way communication is indicated. The receiver can accept and reply to incoming messages. Messages are read via either the READUPDATE or the READ procedure. Messages read via READUPDATE must be replied to via the REPLY procedure; messages read via READ are not replied to.

If <receive depth> > 1, message queueing is indicated. The maximum number of messages that the application process expects to have queued at any given moment must be specified in the <receive depth> parameter. If message queueing is performed, then a message tag must be obtained in a call to the LASTRECEIVE procedure immediately following each call to READUPDATE and passed to the REPLY procedure when replying to the message.

PROCESS FILES

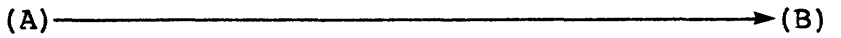
A process ID is used to open a file and to send messages to a designated process and, optionally, wait for a reply.

A process ID has two forms:

If it references a process not in the Process-Pair Directory (PPD), it consists of

```
<process id>[0:2] = <creation time stamp>
<process id>[3]   = <cpu,pin>
```

which is assigned by the GUARDIAN operating system at process creation time. If this form of process ID is used to open a file, the file references that process explicitly:



A opens B using B's process ID. Communication occurs explicitly with B. If B stops or if B's processor module fails, communication can no longer occur.

If the process ID references a process or a pair of processes whose name is in the PPD, it consists of

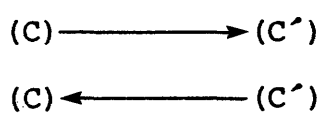
```
<process id>[0:2] = $<process name>
<process id>[3]   = "  " (two blanks) or <cpu,pin>
```

If the process accessing the pair is a member of the pair, then the process name references the opposite member of the pair. The <sync depth> parameter is ignored in this case.

For example, each member of the process pair "$SERVE" opens a file using the process name "$SERVE":

```
INT .fname[0:11] := ["$SERVE", 9 *["  "]];

.
CALL OPEN ( fname, fnum );
```



C opens the pair using the process name $SERVE. C communicates with its backup process C´. Likewise, C´ opens the pair using the process name $SERVE. C´ then communicates with its primary process C.

The process name form of the process ID can be further qualified at file open time by the addition of one or two optional qualifier names. This provides for process file names of the form:

```
word:
[0:3]                [4:7]                  [8:11]
$<process name> [ [ #<1st qualif name> [ <2nd qualif name> ] ]
```

where

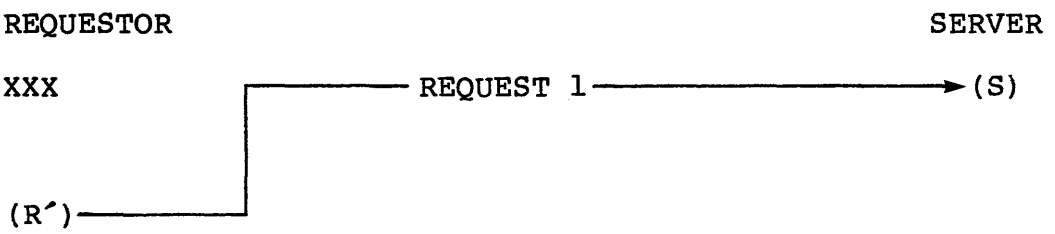
#<1st qualif name>

consists of a number sign "#" followed by one to seven alphanumeric characters, the first of which must be alphabetical.

<2nd qualif name>

consists of one to eight alphanumeric characters, the first of which must be alphabetical.

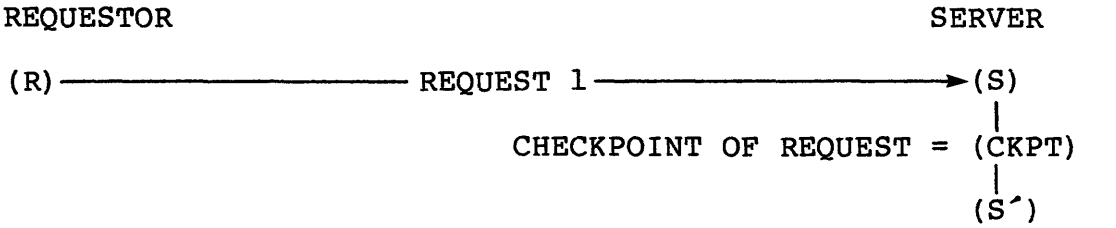
Failure of primary:



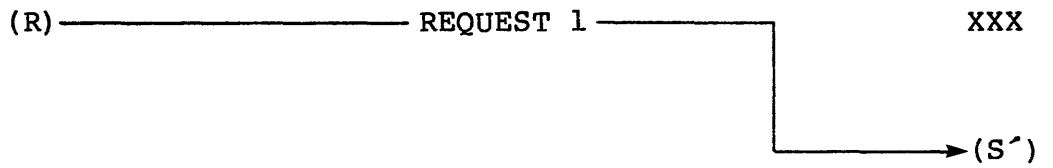
This results in the identical request being sent to the server. The server must recognize the request as a duplicate and return the latest reply for the requestor.

2. By the file system reexecuting the latest request from a requestor process because the primary server process failed:

Normal:



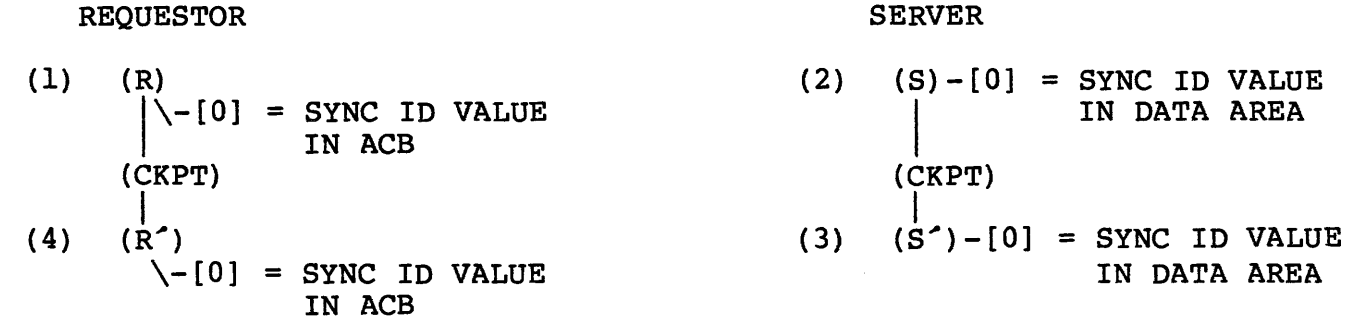
Failure of primary server (causing the file system to reexecute the request to (S') on behalf of (A)):



The backup server (S') may have executed the request on its takeover from (S). If so, the backup server must identify the request as being one it has already executed and return the appropriate reply to the requestor.

Each process file that is open has its own sync ID. A sync ID is a double-word, unsigned integer that is kept in a process file's ACB. Sync ID's are not part of the message data; rather, the sync ID value associated with a particular message is obtained by the receiver of a message by calling the RECEIVEINFO procedure (the receiver must keep the sync ID value associated with a message in its data area).

A file's sync ID is set to zero at file open and when the RESETSYNC procedure is called for that file (RESETSYNC can be called directly and is called indirectly by the CHECKMONITOR procedure; see section 5, "Checkpointing Facility"). When RESETSYNC is called for a process file, a RESETSYNC system message is sent to that process file. The receipt of the message allows the process to clear its copy of the



Successful Transaction:

1. Transaction begins. The requestor builds the request message, then checkpoints the request message and current sync ID value (i.e., 0) to the backup requestor.

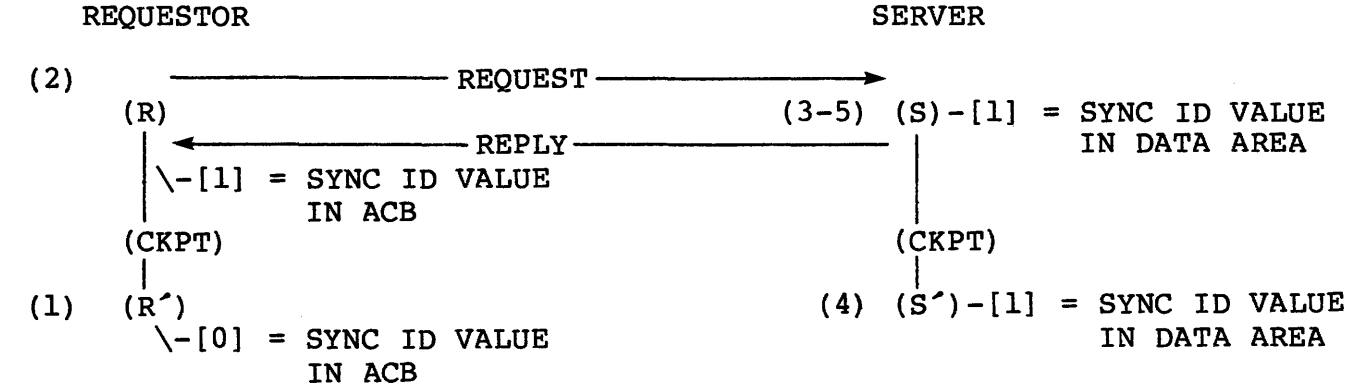
2. The requestor sends the request message to the server. At this time the file system increments the sync ID value by 1.

3. The server picks up the request via $RECEIVE, then calls RECEIVEINFO to obtain the sync ID value.

4. The server examines the sync ID value for the requestor to determine if it matches a request it has already received. If it does not, the server checkpoints the request and the sync ID value to the backup server (S'), executes the request, and saves the reply value for the request.

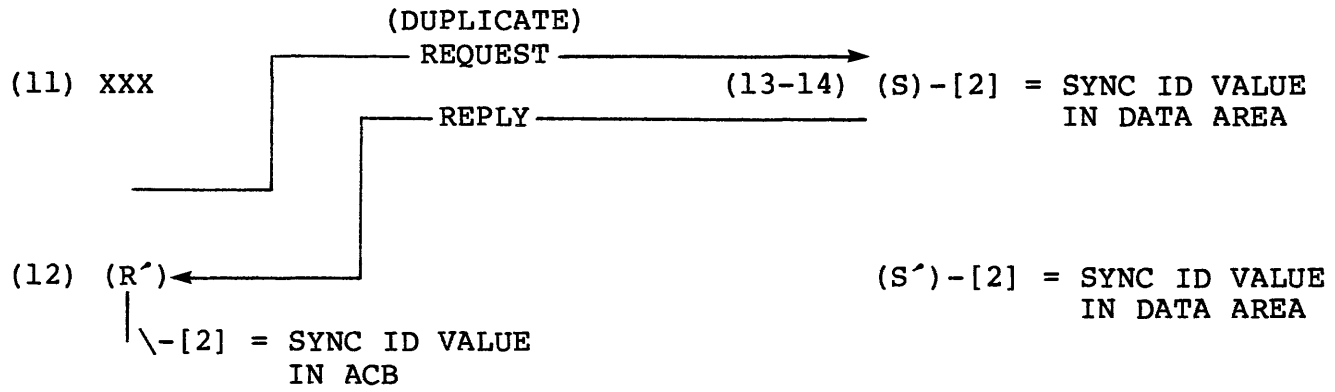
   If the request's sync ID does match, the saved reply value is returned to the requestor.

5. The server then returns the reply value to the requestor via a call to REPLY.



Transaction with failure of requestor primary (sync ID = 1):

6. Transaction begins. The requestor builds the request message, then checkpoints the request message and current sync ID value (i.e., 1) to the backup requestor.



Transaction with failure of server primary (sync ID = 1):

6. Transaction begins. The requestor builds the request message, then checkpoints the request message and current sync ID value (i.e., 1) to the backup requestor.

7. The requestor sends the request message to the server. At this time, the file system increments the sync ID value by 1. The sync ID value is now 2.

8. The server picks up the request via $RECEIVE, then calls RECEIVEINFO to obtain the sync ID value.

9. The server checkpoints the request and the sync ID value to the backup server (S'), executes the request, and saves the reply value for the request.

10. The server primary fails (note that this example is valid no matter when the primary server may fail).

11. The backup server takes over and becomes the primary server. It executes the latest request that was checkpointed by the failed primary. The new primary server then attempts to reply to the request, but because there is no actual request pending for this process, the reply fails (the failure is ignored).

12. The file system, on behalf of the requestor process, sends the current request to the new primary server.

13. The server picks up the request via $RECEIVE, then calls RECEIVEINFO to obtain the sync ID value.

14. The server examines the sync ID value for the requestor to determine if it matches a request it has already received. Because the sync ID does match the sync ID for a request from this requestor, the server knows that it has already executed this operation. Therefore, it returns the appropriate "saved" reply value for this request.

INTERPROCESS COMMUNICATION EXAMPLE

The following is an example of a two-way transmission between a requestor process and a server process. The server accepts OPEN and CLOSE system messages and rejects CONTROL, SETMODE, and CONTROLBUF system messages. No message queueing is performed. Only one open is permitted per requestor process.

The following depicts the call in the requestor process to open the server process:

```
INT .sfname[0:11] := ["$SERVE",9 * ["  "]];

CALL OPEN ( sfname, sfnum,, 1 );
```

opens a file to the server process. Automatic path error recovery is specified.

The following depicts the calls in the server process to initialize the $RECEIVE file:

```
INT .recv^fname[0:11] := ["$RECEIVE",8 * ["  "]];

LITERAL flags = %40001,  ! enable OPEN, CONTROL, etc. system
                         ! messages, no-wait i/o.
        recv^depth = 1;  ! reply used; no message queueing.

CALL OPEN ( recv^fname, recv^fnum, flags, recv^depth );
```

opens the $RECEIVE file in the server process.

The server also calls the MONITORCPUS procedure. This is done so that it will be informed if failure occurs in a processor module of any process it is serving (see "Checkpointing Facility" for a description of MONITORCPUS):

```
CALL MONITORCPUS ( -1 );
```

monitors all processor modules in the system.

The following depicts the action of the server process when reading the $RECEIVE file:

```
INT .recv^buf[0:255], ! receive buffer.
     recv^cnt,        ! receive count.
    .pid[0:3],        ! requestor <process id>.
     system^message;  ! state flag.
INT(32) sync^id,      ! request sync ID value.

WHILE 1 DO ! loop on requests.
  BEGIN
    CALL READUPDATE ( recv^fnum, recv^buf, 512 ); ! $RECEIVE.
    CALL AWAITIO ( recv^fnum,, recv^cnt );
    IF >= THEN ! read a message.
```

```
    ! -32 ! ! CONTROL system message.
            reply^error^code := 2; ! invalid operation.

    ! -33 ! ! SETMODE system message.
            reply^error^code := 2; ! invalid operation.

    ! -34 ! ! RESETSYNC system message.
            BEGIN
              requestor := lookuppid ( pid );

                The "lookuppid" procedure is used to look up a
                requestor process sync in a local directory.  If the
                "pid" exists, the entry number in the directory is
                returned.  If not, a zero is returned.

              sync^count [ requestor ] := 0D;
            END;

    ! -35 ! ! CONTROLBUF system message.
            reply^error^code := 2; ! invalid operation.

            OTHERWISE ! other system message.
              BEGIN
                ! check for CPU Down message.
                IF recv^buf = -2 THEN CALL delallpids ( recv^buf[1] )

                   The "delallpids" procedure is used to delete all
                   processes from the local directory associated with
                   the processor module that failed.

                ELSE
                   .
                   .
              END;
          END; ! system message case.

          ! reply to system message.
          CALL REPLY (,,,, reply^error^code);

      END; ! process^system^message.
```

The following depicts the action of the requestor process to send a request and wait for a reply from the server process.

```
    .
  WHILE 1 DO
    BEGIN
        .
        .  A request is generated by the occurrence of an external
        .  event.
        .
      ! format and send request message to "server".
      send^buffer ':=' request FOR request^len;
```

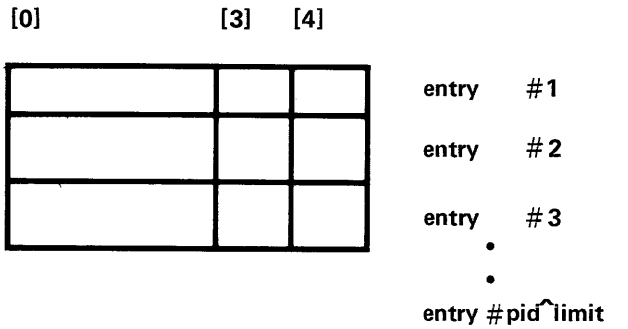
```
    ! return the reply to the requestor.
    CALL REPLY ( reply^buf [ requestor * reply^size ],
                 reply^len [ requestor ],
                 ,
                 ,
                 reply^error^code [ requestor ] );

       If this is a duplicate request, the last reply is returned
       to the requestor.

  END; ! process^user^request.
```

The following are the procedures in the server process that maintain the local directory of process ID's. The directory is of the form

```
INT .pids[5:pid^limit * 5 + 5] := (pid^limit * 5) * [ 0 ];
```



entry [0:2] = process name OR creation time stamp
entry [3] = cpu, pin OF PRIMARY PROCESS
entry [4] = cpu, pin OF BACKUP PROCESS, IF ANY, OR ZERO

```
INT PROC lookuppid(pid);
   INT .pid;

  ! return values:
  !  0 = PID not in directory.
  ! >0 = entry no. of PID in directory.

  BEGIN
    INT entryno := 0, ! entry no. in local PID directory.
        comp^len;     ! compare length for PID matching.

    comp^len := IF pid.<0:7> = "$" THEN ! process name ! 3 ELSE 4;
    WHILE (entryno := entryno + 1) <= pid^limit DO
      IF pid = pids[entryno * 5] FOR comp^len THEN ! found it.
        RETURN entryno;

    RETURN 0; ! not found.
  END; ! lookuppid.
```

```
PROC delallpids(cpu);
   INT cpu; ! processor module number of PID's to be deleted.

  BEGIN
    INT entryno := 0, ! entry in local PID directory.
        temp;

    WHILE (entryno := entryno + 1) <= pid^limit DO
      BEGIN ! check each entry.

        ! check for match with entry's primary cpu.
        IF pids[ entryno * 5 + 3 ] AND
           pids[ entryno * 5 + 3 ].<0:7> = cpu THEN ! primary down
          ! delete primary process and maybe the entire entry.
          CALL delpid ( pids [ entryno * 5 ] )
        ELSE
        ! check for match with entry's backup cpu.
        IF pids[ entryno * 5 + 4 ] AND
           pids[ entryno * 5 + 4 ].<0:7> = cpu THEN ! backup down.
          ! clear the backup entry.
          pids[ entryno * 5 + 4 ] := 0;
    END;
END; ! delallpids.
```

SYSTEM MESSAGES

The following messages from the operating system may be sent to an application process through the $RECEIVE file.

The first word of a system message always has a value less than zero. Also, the completion of a read associated with a system message returns a condition code of CCG (greater than) and error 6 from FILEINFO.

Note: Like all interprocess messages, system messages read via calls to the READUPDATE procedure must be replied to in a corresponding call to REPLY. If the application process is performing message queueing, LASTRECEIVE or RECEIVEINFO must also be called immediately following the completion of the READUPDATE, and the message tag must be passed back to the REPLY procedure.

The system messages and their formats, in word elements, are as follows:

- CPU Down Message. There are two forms of the CPU Down message:

  ```
  <sysmsg>          = -2
  <sysmsg>[1]       = <cpu>
  ```

  This form is received if a failure occurs with a processor module being monitored. Monitoring for specific processor modules is requested by a call to the process control MONITORCPUS procedure.

This form is received by a deleted process's creator if the deleted process was not named or by one member of a process pair when the other member is deleted.

```
<sysmsg>              = -6
<sysmsg>[1] FOR 3     = $<process name> of deleted process [pair]
<sysmsg>[4]           = -1
```

This form is received by a process pair's ancestor when the process name is deleted from the PPD. This indicates that neither member of the process pair exists.

- Change in Status of Network Nodes

```
<sysmsg>              = - 8
<sysmsg>[1].<0:7>     = system number
<sysmsg>[1>.<8:15>    = number of cpu's
<sysmsg>[2]           = new processor status bitmask
<sysmsg>[3]           = previous processor status bitmask
```

This message is received if the process is running on a system that is part of a network, and has enabled receipt of remote status change messages by passing "1" as a parameter to the MONITORNET procedure.

- SETTIME Message (NonStop II systems only)

```
<sysmsg>              = -10
<sysmsg>[1]           = <cpu>
```

This message is received if the interval clock of <cpu> has been reset by a the system manager or operator, provided the process has enabled receipt of new messages by a call to MONITORNEW.

- Power On Message (NonStop II systems only)

```
<sysmsg>              = -11
<sysmsg>[1]           = <cpu>
```

This message is received if the indicated processor had a POWER OFF, then a POWER ON condition, provided the process has enabled receipt of new messages by a call to MONITORNEW.

- NEWPROCESSNOWAIT Completion Message (NonStop II systems only)

```
<sysmsg>              = -12
<sysmsg>[1]           = <error>
<sysmsg>[2] FOR 2     = <tag>
<sysmsg>[4] FOR 4     = <process id>
```

This message is received by a process when a call to the NEWPROCESSNOWAIT procedure is completed.

Receipt of the following six system messages (OPEN, CLOSE, CONTROL, SETMODE, RESETSYNC, and CONTROLBUF) is possible only if the process has opened its $RECEIVE file with <flags>.<1> = 1:

- Process OPEN Message

<sysmsg>	= -30
<sysmsg>[1]	= <flags> parameter to caller's OPEN
<sysmsg>[2]	= <sync or receive depth> parameter to caller's OPEN
<sysmsg>[3] FOR 4	= 0 if normal open, <process id> of primary process if an open by a backup process
<sysmsg>[7]	= 0 if normal open, <file number> of file if an open by a backup process
<sysmsg>[8]	= <process accessor id> of opener
<sysmsg>[9] FOR 4	= optional #<1st qualif name> of named process or blanks
<sysmsg>[13] FOR 4	= optional <2nd qualif name> of named process or blanks

This message is received by a process when it is opened by another process. The process ID of the opener can be obtained in a subsequent call to LASTRECEIVE or RECEIVEINFO.

Note: This message is also received if the open is by the backup process of a process pair. Therefore, a process can expect two of these messages when being opened by a process pair.

- Process CLOSE Message

<sysmsg>	= -31

This message is received by a process when it is closed by another process. The process ID of the closer can be obtained in a subsequent call to LASTRECEIVE or RECEIVEINFO.

Note: This message is also received if the close is by the backup process of a process pair. Therefore, a process can expect two of these messages when being closed by a process pair.

- Process CONTROL Message

<sysmsg>	= -32
<sysmsg>[1]	= <operation> parameter to caller's CONTROL
<sysmsg>[2]	= <parameter> parameter to caller's CONTROL

This message is received when another process calls the CONTROL procedure referencing the receiver process file. The process ID of the caller to CONTROL can be obtained in a subsequent call to LASTRECEIVE or RECEIVEINFO.

ERROR RECOVERY

For the $RECEIVE file, there are no error conditions, with the exception of error 40 (timeout), for which error recovery should be attempted.

For a process file opened with a synchronization depth greater than zero, there are no error conditions, with the exception of error 40, for which error recovery should be retried.

For a process file opened with a synchronization depth of zero, an operation which returns error 201 ("path down") should be retried once if the process file is a process pair. An occurrence of error 201 means that the primary process failed. A reexecution of the call that returned the error causes communication to occur with the backup process, if any. If no backup process exists, a second error 201 will be returned on the reexecution of the call. At this point, the error can be considered fatal.

The operator console is used to log the occurrence of system error conditions, to log system statistical information, and to log application-supplied information. Any process can write messages on the operator console through use of standard file system procedures.

There are three places where console messages may be directed:

1. A console terminal device
   This is typically a hard-copy device for NonStop systems; for NonStop II systems, it may be the Operations and Service Processor (OSP). Console message logging can be redirected to other devices, and may be disabled. (See "Console Messages" in the NonStop System Operations Manual or the NonStop II System Operations Manual.)

2. A disc log file designated $SYSTEM.SYSTEM.OPRLOG
   System-generated console messages (and all other messages, for NonStop II systems) are also logged to a disc file, if disc file logging is enabled. Messages are encoded in a special format; the file's contents can be displayed by means of the Tandem-supplied DUMPLOG program. (See "Console Messages" in the NonStop System Operations Manual or the NonStop II System Operations Manual.)

3. An application process named $AOPR
   If an application process named $AOPR exists, all console messages will be logged to it. The console messages are sent by means of an interprocess message (message -7). A console logging system message contains the exact image of the console message. (See "Console Logging to an Application Process" in this section.)

The following topics are covered in this section:

- General Characteristics of the Operator Console
- Summary of Applicable Procedures
- Writing a Message
- Console Message Format
- Error Recovery
- Console Message Logging to an Application Process

GENERAL CHARACTERISTICS OF THE OPERATOR CONSOLE

- The operator console is accessed by

  $0 (dollar sign, zero).

- It is opened like any other file (including no-wait).

- Maximum message length is the console device record length minus 30 bytes (102 bytes); longer messages are truncated.

- The operator console is a write-only device.

- Operator messages preempt terminal reads on the operator console.

The message appears on the console as follows:

```
14:30 07SEP75 FROM 4,24 LOAD TAPE NO. 12345 ON UNIT 2
```

To terminate access to the operator console, the CLOSE procedure is used:

```
    .
CALL CLOSE ( op^file^num );
    .
```

CONSOLE MESSAGE FORMAT

The general form of console messages is:

```
<time stamp> FROM <cpu>,<pin> <message>
```

where

<time stamp> is the current date and time of day.

<cpu> is the number of the cpu where the process that sent the message is executing.

<pin> is the process information number associated with the execution.

<message> was transmitted to the operator from an application program and begins in column 31 on the console device.

ERROR RECOVERY

The file system automatically retries path errors to the operator console (i.e., errors numbered 200 or greater); therefore, the application program can consider these to be permanent errors.

The application program, however, should take care of errors associated with device operation; these are errors such as "not ready" or "paper out".

CONSOLE LOGGING TO AN APPLICATION PROCESS

All console messages (both system- and application-generated) are logged to an application process named $AOPR if such a process exists. The message is sent by the operator process as an interprocess message. The application process reads the message via its $RECEIVE file (see "Interprocess Communication").

The features described in this section are considered "advanced features" because, if misused, they could result in a degradation of system performance.

The advanced features covered in this section are:

- Reserved Link Control Blocks
- Resident Buffering

RESERVED LINK CONTROL BLOCKS

A Link Control Block (LCB) is a system resource that is used when a message is sent from one process to another (figure 2-25). An LCB contains control information about the message. Before a message transfer can take place, an LCB must be secured on the sender's side (a "send" LCB) and another LCB (a "receive" LCB) must be secured on the receiver's side. This means that a pair of LCB's is required for each message transfer that is in progress at any given moment. Note also that a call to most file system and process control procedures results in a message being transferred between the calling application process and a system process.

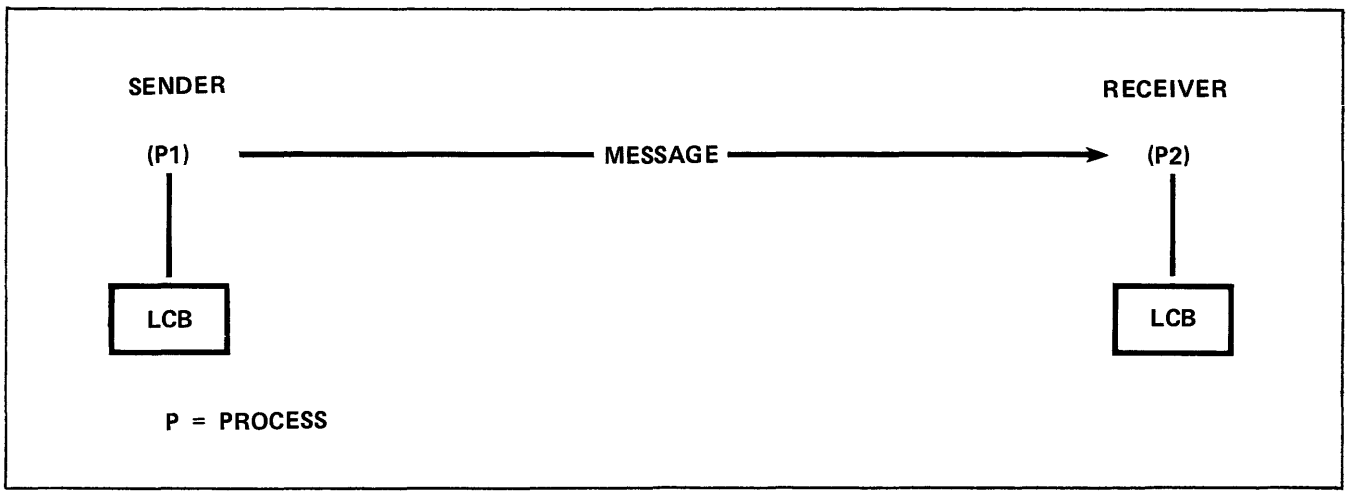


Figure 2-25. Link Control Blocks

The reserved Link Control Blocks feature is used so that an application process will not be suspended, waiting for an LCB to be allocated.

An application process can reserve Link Control Blocks by calling the RESERVELCBS procedure. This requires specifying the number of LCB's to be reserved for receiving messages (i.e., receive LCB's) and the number to be reserved for sending messages (i.e., send LCB's):

The call to the RESERVELCBS procedure is:

```
CALL RESERVELCBS ( <no. receive lcbs> , <no. send lcbs> )
---- ----------- - ----------------- - --------------- -

where

   <no. receive lcbs>, INT:value,

     specifies the number of "receive" LCB's to be reserved for
     this process: {0:255}.

   <no. send lcbs>, INT:value,

     specifies the number of "send" LCB's to be reserved for this
     process: {0:255}.

condition code settings:

   <  (CCL) indicates that not enough unreserved LCB's are
            available to reserve the LCB's specified in this call,
            or the amount requested by either parameter is not in
            the range of {0:255}.  The number of reserved LCB's
            allocated to this process is unchanged.
   =  (CCE) the requested LCB's are reserved for this process.
   >  (CCG) is not returned by RESERVELCBS.

example:

   CALL RESERVELCBS ( 1, 4 );
   IF < THEN ... ; ! failed.
```

CONSIDERATIONS

- A process may call RESERVELCBS multiple times; the latest call is the one that is used.
- The worst-case values for <no. receive lcbs> and <no. send lcbs> can be calculated as follows:

```
<no. receive lcbs> =   1  for each terminal where break is being
                           monitored
                     + 1  for each cpu being monitored (i.e., by
                           MONITORCPUS)
                     + 1  for each process of which this process is
                           the creator or ancestor
                     + 1  for each possible interprocess message
                           from application processes.
```

RESIDENT BUFFERING (NonStop systems only)

When resident buffering is used (figure 2-26), the data transferred because of an i/o request is transferred directly between the application process's data area and an i/o buffer in the processor module where the primary i/o process controlling a device is located. This bypasses the normal intermediate transfer to a file system buffer in the processor module where the application process is running. In addition to saving a move operation, using resident buffering also means that an application process will not be suspended, waiting for file system buffer space to become available, when performing an i/o operation.

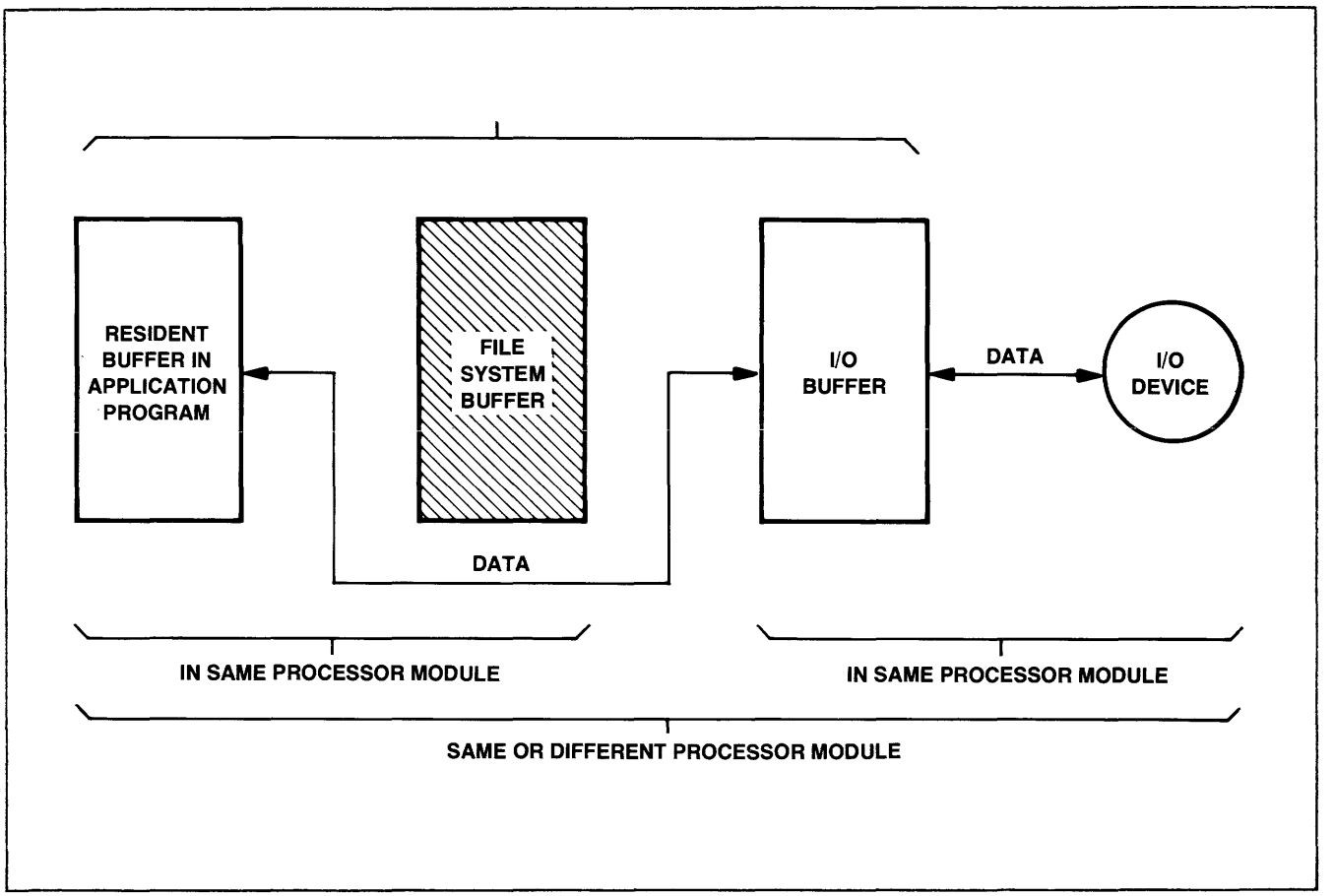


Figure 2-26. Resident Buffering (NonStop systems only)

Resident buffers are specified on a file-by-file basis through <flags>.<6> of the OPEN procedure. If resident buffers are specified, the application process must make any buffers (i.e., arrays) used with the file main-memory resident. Additionally, the resident buffer in the application's data area must be addressable through the system data map. Both are done through a call to the LOCKDATA procedure described in section 8. LOCKDATA can be called only if the application process is executing in privileged mode; otherwise an

Then the file is opened. Resident buffering is specified as shown in the previous call to OPEN.

CONSIDERATIONS

- If resident buffering is to be used, at least 1K of system global data space must be left unassigned when generating the system with SYSGEN.
- Although resident buffering is specified on a file-by-file basis, a process may use the same resident buffer for several different files (if, of course, the structure of the program permits).
- It is not necessary to call LOCKDATA before OPEN is called. However, LOCKDATA must be called before the first i/o transfer (i.e., READ, WRITE, CONTROL, etc.) with a file is performed.
- The resident buffer is not used for accesses to structured ENSCRIBE files.
- For further information, see "Considerations" for the LOCKDATA procedure in section 8.

SECTION 3

PROCESS CONTROL

This section provides a general overview of the following:

- Process Definition
- Process States
- Process ID
- Creator
- Process Pairs
- Named Processes (Process-Pair Directory):
  - Primary Process
  - Backup Process
  - Operation of the PPD
  - Ancestor Process
- Home Terminal
- Elapsed Timeout (NonStop II systems only)

PROCESS DEFINITION

A "process" is the basic work unit of the GUARDIAN operating system. A program (either an application or system program) running on the system is a process. Specifically, the term "program" indicates a static group of instruction codes and initialized data (like the output of a compiler), whereas the term "process" identifies the dynamically changing states of an executing program. The same program can be executing concurrently a number of times; each execution is considered a separate process. (See figure 3-1.)

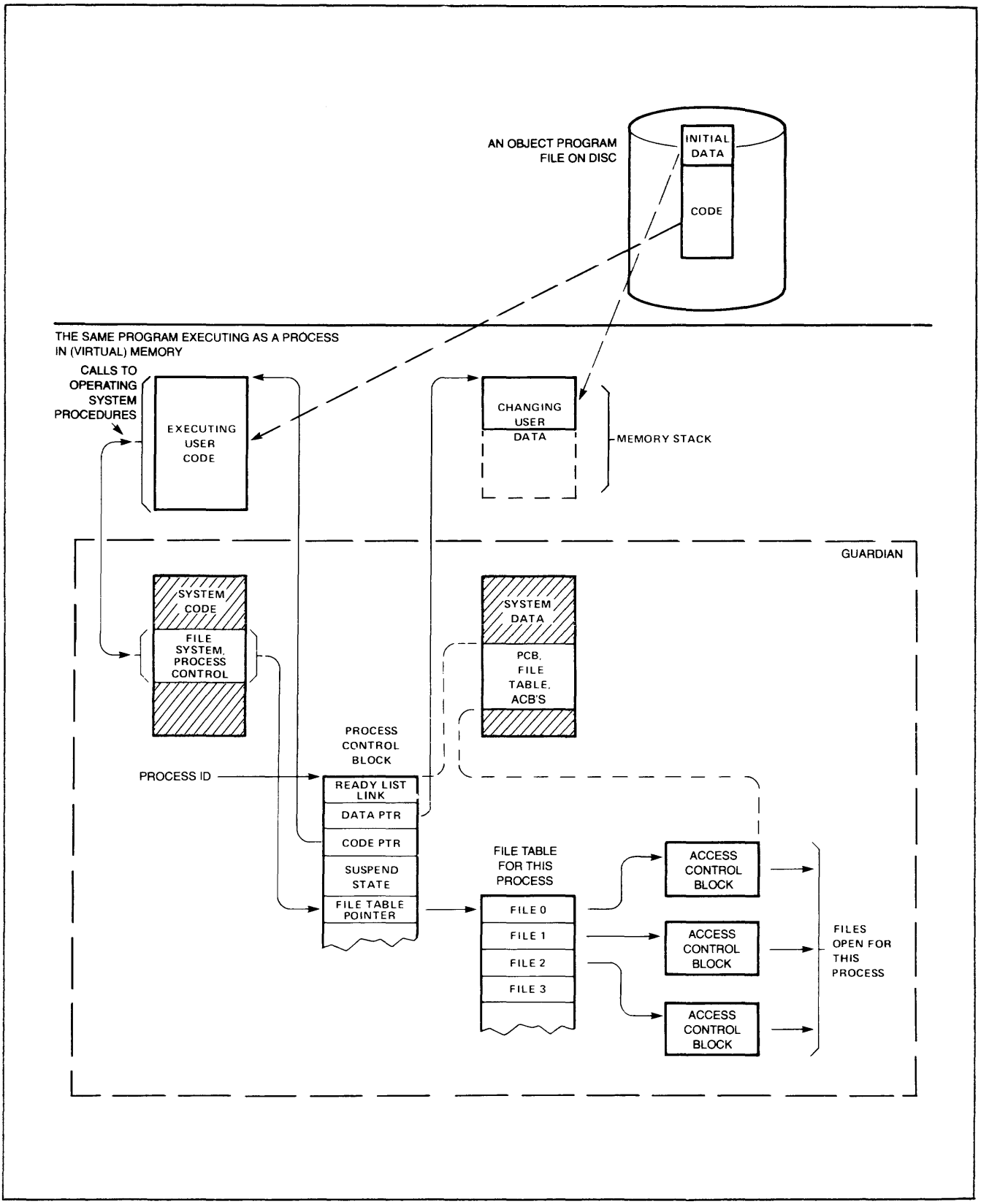


Figure 3-2. A Process (NonStop systems)

PCB contains pointers to the process's code and data areas (real and virtual), retains the current state of the process when the process is suspended, and contains pointers to files open by the process.

PROCESS STATES

During its existence, a process goes through the following states:

CREATION ——→ EXECUTION ——→ DELETION

Creation

The term "creation" refers to the action performed by a system process called the System Monitor when a program is initially prepared for execution. During process creation, the System Monitor performs a number of operations. Some of these are:

- Locating the program file on disc
- Assigning a process ID
- Allocating and initializing a PCB
- Determining if the code part is being executed by another process (for code sharing)
- Allocating space for copies of the data and (if not sharing code) code maps
- Allocating virtual memory space. For code, the program file is used as the virtual area; for data, space on the same volume as the program file is used as the virtual area.
- Linking references to external procedures (e.g., calls to file system procedures) in the application program file to the operating system. This linking is necessary only the first time the program is run. Subsequent process creations with the same program file skip this step.
- If the process is named, an entry is made for the process into the Process-Pair Directory (PPD). (See "Named Processes").

Process creation is initiated by either application programs or the Command Interpreter (COMINT) through the process control procedure NEWPROCESS or NEWPROCESSNOWAIT. (NEWPROCESSNOWAIT is available only on NonStop II systems.) The processor module where the program is to execute is specified (any module is permitted) along with its execution priority and the maximum number of data pages permitted.

To protect the system against excessive loss of throughput due to a process that (often due to program errors) is extremely cpu-bound, the GUARDIAN operating system -- on NonStop II systems only -- includes a "floating priority" feature. If a single process retains uninterrupted control of a cpu for a given length of time (determined at system generation) and other processes of equal or lower priority are thus prevented from running, the operating system will automatically reduce the priority of the procedure so that other processes may run. Each time the process runs uninterrupted for more than the given time, its priority is reduced by one and timing begins again, so that the process's priority is reduced in a stepwise manner. The PRIORITY procedure can be used to check whether such reduction of priority has occurred.

The following process control procedures are related to process execution:

ACTIVATEPROCESS
: is used to return a process that is in the suspended state to the ready state

ALTERPRIORITY
: is used to alter the execution priority of another process

DELAY
: permits a process to suspend itself for a timed interval

PRIORITY
: permits a process to dynamically change its own execution priority

SETLOOPTIMER
: is used to detect a looping process (i.e., executing, but not as expected). SETLOOPTIMER permits a process to set a limit on the total amount of processor time it is allowed (i.e., total time that the process is in the active state). If the time limit is reached, a "process loop timer timeout" trap occurs

SUSPENDPROCESS
: is used to put another process into the suspended state

Deletion

"Deletion" is the act, by the operating system, of stopping further process execution. The deleted process is removed from its current execution state (i.e., active, ready, or suspended), files it has opened are closed, its associated resources (e.g., PCB, memory stack space, code space if not shared) are returned to the system, and the deleted process's creator is notified (by means of a system message) of the deletion.

There are two types of process deletion, normal and abnormal:

where

<process name> must be preceded by a dollar sign "$" and consist of a maximum of five alphanumeric characters; the first character must be alphabetic.

- Process Name Form (Network)

  For named processes in a network, the form of a process ID is

```
<process id>[0].<0:7>      = ASCII "\" (octal 134)
<process id>[0].<8:15>     = system number (in octal)
<process id>[1:2]          = <process name>
<process id>[3].<0:7>      = <cpu>
<process id>[3].<8:15>     = <pin>
```

  Note that <process name> in words 1 and 2 does not include the initial dollar sign "$".

The following process control procedures are related to process ID's:

MYPID provides a process with its own <cpu,pin>

GETCRTPID provides the process ID associated with a <cpu,pin>

GETREMOTECRTPID

provides the process ID associated with a <cpu,pin> in a remote system

CREATOR

The term "creator" refers to the relationship that exists between a process that initiated a process creation (i.e., the caller to NEWPROCESS) and the process that was created.

For example, the Command Interpreter is the creator of the process created when a RUN command is given:

```
:RUN myprog
```

command is given to run a program.

(CI) creator, Command Interpreter

(A) process created due to RUN command

The purpose of the creator relationship is to designate the process to be notified when a process is deleted (the notification is in the form of a Process STOP or Process ABEND system message):

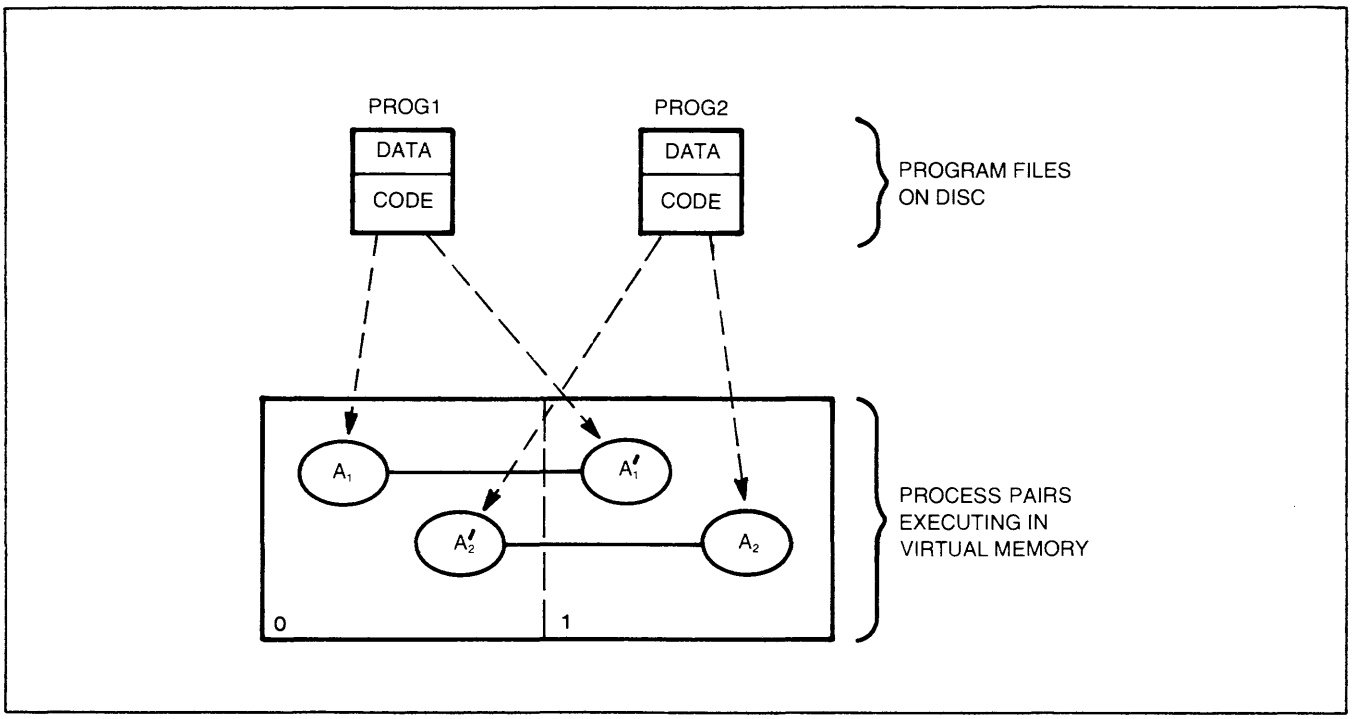


Figure 3-4. Process Pairs

One process of the pair is designated the primary; the other is designated the backup. Logic in the program indicates whether the process is executing in the primary mode or the backup mode.

```
(A)————————(A')

PRIMARY        BACKUP
```

During a process pair's existence, it passes through the following process execution states:

```
PRIMARY

   CREATION———EXECUTION———►DELETION OR FAILURE
                  .
                  .
                primary creates backup
                  .
BACKUP            .
                CREATION———EXECUTION———►DELETION OR FAILURE
                                .
                                .
                              if primary fails, backup becomes
                              primary and creates a new backup.
                                .
BACKUP                          .
                              CREATION———►.....etc.
```

entry#	word [0:2]	[3]	[4]	[5:8]
[0]	$<process name>	<cpu,pin 1>	<cpu,pin 2>	<ancestor process id>
[1]	.	.	.	.
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.
[n-1]	.	.	.	.

Each entry consists of a process name, the <cpu,pin>s of the two processes comprising the pair, and the process ID of the process or process pair responsible for creation of the primary.

A process name is entered into the PPD at process creation time via a parameter to the NEWPROCESS or NEWPROCESSNOWAIT procedure. Any process may create a process and assign it an unused process name. Only a primary process, however, may create the second process (i.e., the backup) associated with its name. A process can have the system generate, via a call to the CREATEPROCESSNAME Procedure, a previously undefined, and unique, process name. The system-generated process name is used when a process pair need not be known to other processes, but the fault-tolerant aspects of named processes are desired. (A process name, either predefined or system-generated, can be assigned via the Command Interpreter RUN Command.)

When two processes are associated with a name, the two processes become each other's "creator". One process is notified of the other's deletion; each process can stop the other.

($N:C,P 1) ←——— EACH OTHER'S CREATOR ———→ ($N:C,P 2)

($N:C,P #) represents a member of a named process pair
(N = process name: C = <cpu>, P = <pin>)

When a process represented in the PPD by a given name is deleted or its processor module fails, the reference to the particular process (i.e., its <cpu,pin>) is zeroed in the PPD, and the other process (if any) becomes the primary. When the new primary creates a new process having that name, the new process's <cpu,pin> is entered into the PPD.

When the last process associated with a given name is deleted, the process name is deleted, and the ancestor of the process pair (if alive) is notified. The deleted process name can then be reused.

Ancestor Process

The "ancestor" relationship can exist between the following:

- A non-named process and a named process pair

- Process abnormal deletion (ABEND) message:

```
<sysmsg>              = -6
<sysmsg>[1] FOR 3     = $<process name> of deleted process [pair]
<sysmsg>[4]           = -1
```

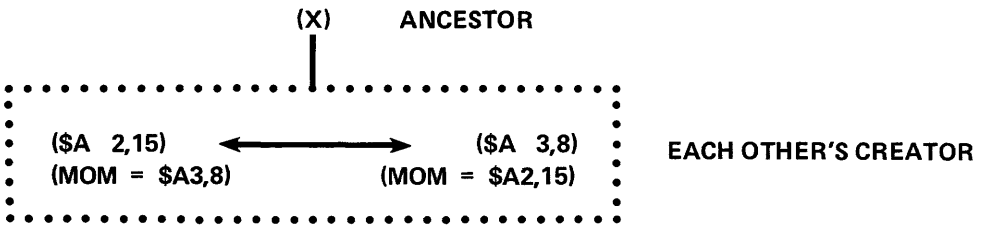
  This message is received if the deletion is due to a call to the process control ABEND procedure, or because the deleted process encountered a trap condition and was aborted by the operating system. It is received by a process pair's ancestor when the process name is deleted from the PPD. This indicates that neither member of the process pair exists.

Note: If the ancestor process responsible for the original primary's creation is a member of a process pair, the ancestor process pair receives this notification regardless of whether or not the actual creating process still exists.

5. ($A 3,8) creates its backup:

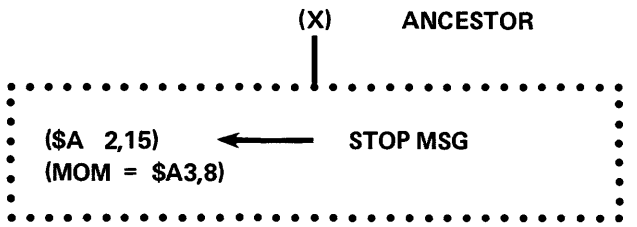
```
name ':=' "$A     ";
CALL NEWPROCESS(...,name);
```

$A	3,8	2,15	X



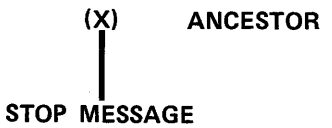
6. ($A 3,8) stops:

$A	2,15	0	X



7. ($A 2,15) stops:

DELETED



Procedures

The following procedures are used for performing operations involving named process [pairs]:

CREATEPROCESSNAME

is used to have the system generate a process name suitable for passing to the NEWPROCESS or NEWPROCESSNOWAIT procedure

LOOKUPPROCESSNAME

returns the PPD entry associated with a process name

NEWPROCESS creates a new process (runs a program) and, optionally, enters its application-defined symbolic process name into the PPD

ELAPSED TIMEOUT (NonStop II systems only)

The NonStop II system allows a process to set timers that count actual elapsed time. (This differs from process time, or run time, as counted by the DELAY procedure.) When the set time interval for a timer has expired, the time list entry allocated for the timer is queued on the process' $RECEIVE queue ahead of any other type of request, but behind any other time list entry. The user process sees the timeout as a system message (file system error 6, CCG) with the following format:

```
<sysmsg>                = -22
<sysmsg>[1]             = <parameter1> supplied to SIGNALTIMEOUT
                              (if none supplied, 0)
<sysmsg>[2] FOR 2       = <parameter2> supplied to SIGNALTIMEOUT
                                (if none supplied, 0D)
```

A process may do exact elapsed timings by using the RCLK instruction.

The following procedures are related to the elapsed timeout feature:

SIGNALTIMEOUT	sets a timer for a given period of elapsed time
CANCELTIMEOUT	cancels a timer previously set by SIGNALTIMEOUT

The process control procedures are:

ABEND — deletes the calling process and flags the deletion the result of an abnormal condition

ACTIVATEPROCESS — is used to return a process that is in the suspended state to the ready state

ALTERPRIORITY — is used to alter the execution priority of another process

CANCELTIMEOUT (NonStop II systems only)

cancels an elapsed-time timer previously set by SIGNALTIMEOUT

CONVERTPROCESSNAME

converts a process name from local to network form

CREATEPROCESSNAME

is used to have the system generate a process name suitable for passing to the NEWPROCESS or NEWPROCESSNOWAIT procedure

CREATEREMOTENAME

is used to have the system generate a process name for a remote system

DELAY — permits a process to suspend itself for a timed interval

GETCRTPID — provides the process ID associated with a <cpu,pin>

GETPPDENTRY — returns the PPD entry in a remote system associated with an entry number (a number specifying an ordinal position in the PPD)

GETREMOTECRTPID

provides the process ID associated with a <cpu,pin> in a remote system

LOOKUPPROCESSNAME

returns the PPD entry associated with a process name. A PPD entry consists of a process name, the two <cpu,pin>s of the process pair, and the ancestor's process ID

MOM — provides a process with the process ID of its creator

The ABEND procedure is used to delete the calling process and signal that the deletion was because of an abnormal condition (i.e., send an ABEND system message to the deleted process's creator).

When ABEND executes, all open files associated with the deleted process are automatically closed, and if the process had BREAK enabled, BREAK is disabled.

The call to the ABEND procedure is:

```
CALL ABEND
---- -----

example:

   CALL ABEND;       ! delete me.
```

CONSIDERATIONS

- The creator of the aborted process is sent a process abnormal deletion (ABEND) system message (i.e., system message -6) indicating that the deletion occurred. See "Interprocess Communication", section 2.9, for the format of the message.

The ALTERPRIORITY procedure is used to change the execution priority of a process [pair]. ALTERPRIORITY changes the assigned priority initially given to the process and sets the current priority equal to the assigned priority.

The call to the ALTERPRIORITY procedure is:

```
CALL ALTERPRIORITY ( <process id> , <priority> )
---- ------------- - ------------ - ---------- -

where

   <process id>, INT:ref,

     is an array containing the process ID of the process whose
     execution priority is to be changed.  If <process id>[0:2]
     references a process pair and <process id>[3] is specified
     as -1, then the call applies to both members of the
     process pair.

   <priority>, INT:value,

     specifies a new execution priority value in the range of
     {1:199} for <process id>.

condition code settings:

   < (CCL)  indicates that ALTERPRIORITY failed, or no process
            designated as <process id> exists.
   = (CCE)  indicates that the priority of the process has
            been altered.

example:

   CALL ALTERPRIORITY ( pid , pri );
   IF < THEN ...  ! "pid" doesn't exist.
```

CONSIDERATIONS

- The caller of ALTERPRIORITY must have the same process accessor ID as the process [pair] whose priority is it is attempting to change (see "Security System" for an explanation of "process accessor ID") or be the super ID.

The CONVERTPROCESSNAME procedure converts a process name from local to network form.

The call to the CONVERTPROCESSNAME procedure is:

```
CALL CONVERTPROCESSNAME ( <process name> )
---- ----------------- - -------------- -
```

where

<process name>, INT:ref:3,

is passed a process name beginning with "$". On return, this buffer contains the internal network form of the process name: "\" in the first byte, the calling process's system number in the second byte, followed by the process name.

If <process name> does not begin with "$", it is left unchanged.

Example

An example of the action of CONVERTPROCESSNAME, assuming that MYSYSTEMNUMBER is 3:

```
name ':=' "$proc";
CALL CONVERTPROCESSNAME ( name );
```

On return from the call, "name" contains

0	\	%3
1	P	R
2	O	C

CONSIDERATIONS

- The CREATEPROCESSNAME procedure is also useful for creating "pseudo-temporary" disc file names. This type of naming is used when a disc file that is created by one process must be accessed by another process, and that process opens the file with exclusive access (i.e., the file cannot be open by the process that created the file). In the normal case with actual temporary files, the file, because it would be open only by a single process, would be purged when closed by the other process.

  For example:

```
INT .temp^fname[0:11] := ["$VOL1 ", 9 * ["  "]];

CALL CREATEPROCESSNAME ( temp^fname[4] );
temp^fname[4].<0:7> := "Z";
temp^fname[8] ':=' temp^fname[4] FOR 4;
CALL CREATE ( temp^fname );
IF < THEN ... ; ! error.
```

- Under GUARDIAN operating system version "D" or later, CREATEPROCESSNAME creates a name of the form Zddd; under version "C", it creates a name of the form Zdddd.

The DELAY procedure permits a process to have itself suspended for a timed interval.

The call to the DELAY procedure is:

```
CALL DELAY ( <time period> )
---- ----- - ------------ -

where

   <time period>, INT(32):value,

     specifies the time period, in .01-second units, that the
     caller of DELAY is to be suspended.

     A value of less than or equal to 0D results in no delay as
     such, but changes this process's execution state from active
     to ready to permit other processes of the same priority to
     run.

example:

   CALL DELAY ( 1000D );                ! suspend for 10 seconds.
```

The GETPPDENTRY procedure returns a particular entry in a specific system's Process-Pair Directory (PPD).

The call to the GETPPDENTRY procedure is:

```
CALL GETPPDENTRY ( <entry number> , <system number>
                 , <ppd entry> )
```

where

<entry number>, INT:value,

specifies which PPD entry to return. The first entry is 0, the second is 1, etc.

<system number>, INT:value,

specifies the system whose PPD is to be searched for the desired entry.

<ppd entry>, INT:ref:9,

returns the nine-word PPD entry specified by the given entry and system numbers. The format of the <ppd entry> is

<ppd entry>[0:2]	=	process name (in local form)
<ppd entry>[3]	=	<cpu,pin> of primary (cpu in high-order eight bits, pin in low-order eight bits)
<ppd entry>[4]	=	<cpu,pin> of backup
<ppd entry>[5:8]	=	<process id> of ancestor, if any

condition code settings:

<	(CCL)	the PPD in the given system could not be accessed.
=	(CCE)	GETPPDENTRY completed successfully.
>	(CCG)	there is no such entry in the PPD.

example:

```
CALL GETPPDENTRY( entry^num, system, ppd^entry )
IF < THEN ...
```

The GETREMOTECRTPID procedure returns the CRTPID, also known as the process ID, of a remote process whose cpu, pin, and system number are known.

The call to the GETREMOTECRTPID procedure is:

```
CALL GETREMOTECRTPID ( <pid> , <process id> , <system number> )
---- -------------- - ----- - ----------- - --------------- -
```

where

<pid>, INT:value,

is the <cpu,pin> of the process whose process ID is to be returned.

<process id>, INT:ref:4,

is an array of four words where GETREMOTECRTPID returns the process ID of <cpu,pin>. If <system number> specifies a remote system, the process ID is in network form; if <system number> specifies the local system, the process ID is in local form. Both forms of process ID are described under "Process ID" in section 3.1.

condition code settings:

<	(CCL)	indicates the GETCRTPID failed for one of the following reasons: no such process exists, or the remote system could not be accessed, or the process has an inaccessible name, consisting of more than 4 characters.
=	(CCE)	indicates that GETREMOTECRTPID was successful.
>	(CCG)	is not returned by GETREMOTECRTPID.

example:

```
CALL GETREMOTECRTPID ( pid, crtpid, sys^num );
IF < THEN ... ! problems
```

```
INT count := -1;
    entry[0:8],
    done := 0;
    .
    .
DO
  BEGIN
    entry := (count := count + 1);
    CALL LOOKUPPROCESSNAME ( entry );
    IF = THEN ... ! do something with "entry".
    ELSE done := 1;
  END
UNTIL done;
```

Network Use of LOOKUPPROCESSNAME

Remote PPD entries can be obtained by passing the process name (in network form) of the process desired. On return, the process name remains in network form.

This is an example of using LOOKUPPROCESSNAME to get the PPD entry for the named process "$proc" running on the system "\detroit":

```
external^name ':=' 17 * [ "  " ]; ! blanks
external^name ':=' "\detroit.$proc";
CALL FNAMEEXPAND( external^name, internal^name, defaults );
  ! converts "\detroit" to its system number
CALL LOOKUPPROCESSNAME( internal^name );
  ! returns the desired PPD entry
```

To obtain remote PPD entries using an <entry number>, the GETPPDENTRY procedure must be used.

- Network consideration:

  If a process's creator is in a remote system, its process ID is returned by MOM in network form. A process can use this fact to determine whether or not it was created locally.

The MYSYSTEMNUMBER procedure provides a process with its own system number.

The call to the MYSYSTEMNUMBER procedure is:

```
<system number> := MYSYSTEMNUMBER
                   --------------

where

   <system number>, INT,

      returns the caller's system number.
```

CONSIDERATIONS

- If the caller is running in a system that is not part of a network, MYSYSTEMNUMBER returns 0. Since 0 is a legal system number, a process wishing to determine whether the system it is running in is part of a network should contain the code

  ```
  CALL GETSYSTEMNAME( MYSYSTEMNUMBER, name );
  ```

  A return of all blanks in "name" indicates that the system is not part of a network.

The NEWPROCESS procedure is used to create and, optionally, assign a symbolic process name to a new process. Additionally, the execution priority of the new process, the number of memory pages allotted the process, and the cpu where the process is to execute may be specified. On NonStop II systems, a run-time library and a swap file may also be specified. When a new process is created, its process ID is returned to the caller.

The call to the NEWPROCESS procedure is:

```
CALL NEWPROCESS ( <filenames>
---- ---------- - -----------
                , <priority>
                , <memory pages>
                , <processor>
                , <process id>
                , <error>
                , <name>          )
                                  -
```

where

<filenames>, INT:ref:12 or INT:ref:36,

is an array containing <program file> (the twelve-word file name of the program to be run) and optionally, for NonStop II systems only, two additional fields. (See "File Names" in the "File Management System" section for file name format.)

The additional NonStop II fields, which are used only if bit 1 of the <priority> parameter is set to 1, are as follows:

<library file>	the twelve-word file name of a run-time library to be used by the program
<swap file>	the twelve-word file name of a file to be used as a swap file

If <library file> is specified, unsatisfied external references are satisfied first from the specified library, then from the system library. If <library file> is not specified but another process has specified a <library file> for that program, the previously specified library is used. If <library file> has not been specified at all for the program, or if the first word of <library file> is all zeroes, no library file is used. If <swap file> is specified and a file of that name exists, that file is used for memory swaps of the user data stack during execution of the process; if no file of that name exists, a file of that name and of the necessary size is created and used for swaps.

→

Both forms of process ID are described under the heading "Process ID" in section 3.1 of this manual.

If no process was created, zero is returned in <process id>.

<error>, INT:ref:1,

is returned an error number indicating the outcome of the process creation attempt, where

<error>.<0:7>

0 = no error, process created
1 = undefined external(s)
2 = no PCB available
3 = file management error, then
<error>.<8:15> = file management error number
4 = unable to allocate map
5 = unable to get virtual disc space (NonStop systems)
swap file error (NonStop II systems)
6 = illegal file format
7 = unlicensed privileged program
8 = process name error, then
<error>.<8:15> = file management error number
9 = library conflict (NonStop II systems only)
10 = unable to communicate with System Monitor process

These errors are explained more fully under "Errors for NEWPROCESS and NEWPROCESSNOWAIT" following the description of the NEWPROCESSNOWAIT procedure.

<name>, INT:ref,

if present, is a name to be given to the new process. It is entered into the Process-Pair Directory (PPD). <name> is of the form

<process id>[0:2] = $<process name>

where

<process name> must be preceded by a dollar sign "$" and consists of a maximum of five alphanumeric characters; the first character must be alphabetic. (If the process is created in a remote system, and it is necessary to be able to access the process, its name should consist of at most four characters.)

→

- The library file for a process can be shared with an arbitrary number of other processes. However, if the program specified by <program file> is already running with another library or no library, a library conflict error (error 9) occurs. All processes running a given program must use the same library.

<priority>, INT:value,

consists of three parts:

<priority>.<0> is the DEBUG bit. If <priority>.<0> = 1, then a code breakpoint is set in the first executable instruction of the program's MAIN procedure.

<priority>.<1> indicates the interpretation of the <filenames> parameter. If <priority>.<1> = 1, the additional fields in <filenames> are used. If <priority>.<1> = 0, these extra fields are ignored.

<priority>.<8:15> is the execution priority assigned to the new process {1:199}. If <priority>.<8:15> = 0, then the priority of the caller of NEWPROCESSNOWAIT is used. If a value greater than 199 is specified, then 199 is used.

If <priority> is omitted, the caller's priority is used.

<memory pages>, INT:value,

specifies the number of 1024-word memory pages to be allotted the new process. If <memory pages> is omitted or is less than the value assigned when the program was compiled (or UPDATEd), then the compilation value is used. In any case, the maximum number of pages permitted is 64.

<processor>, INT:value,

specifies the processor where the new process is to run. If omitted, the new process runs in the same processor as the caller.

<process id>, INT:ref:4,

is unused by NEWPROCESSNOWAIT.

<error>, INT:ref:1,

is returned an error number indicating the initial outcome of the process creation attempt. Only errors that prevented initiation of process creation are reported in this parameter; if process creation was initiated, any subsequent errors are reported in the completion message on $RECEIVE. The error numbers in the <error> parameter and in the <error> field of the completion message are identical to the error numbers for the NEWPROCESS

→

- If NEWPROCESSNOWAIT cannot initiate process creation (for instance, if an invalid cpu number is specified), no message appears on $RECEIVE. The <error> parameter is returned a nonzero value indicating the error.

- Also see "CONSIDERATIONS" for the NEWPROCESS procedure.

6 ILLEGAL FILE FORMAT
<program file> or <library file> failed one of the tests performed by the System Monitor to determine if the file is actually a program (these include checking for a file code of 100). No process is created. Use the FUP INFO command to check the file code (see the GUARDIAN Operating System Command Language and Utilities Manual).

7 UNLICENSED PRIVILEGED PROGRAM
Program file contained procedures having CALLABLE and/or PRIV attributes, but the program file was not licensed to execute in privileged mode and was not being run by the super ID. Program files are licensed by the super ID by means of the FUP LICENSE command (see the GUARDIAN Operating System Command Language and Utilities Manual). Have the super ID user license program.

8 PROCESS NAME ERROR
Process name was invalid. The specific reason can be determined by examining the file management error number in <error>.<8:15>. Common file management errors and their causes are:

file management error	reason
10 file already exists	1. There is one entry for <process name> in the PPD, and the caller does not have that name. (The second entry can only be creation by a primary process having <process name>.)
	2. Attempt was made to create a backup process in the same processor module as the primary.
13 illegal file name	1. <process name> is not in the proper form.
45 file is full	1. There are already two entries for <process name> in the PPD.

No process is created. See "File Management Errors" in section 2.4 for corrective action.

9 LIBRARY CONFLICT (NonStop II systems only)
<library file> was specified, but this program was already running with another library or no library. All processes running a given program must use the same library.

10 UNABLE TO COMMUNICATE WITH SYSTEM MONITOR
Process was unable to communicate with System Monitor, possibly because the processor module where the program was to be run did not exist or was inoperable. No process is created. Select another cpu and try again.

The PROCESSINFO procedure is used to obtain process status information.

The call to the PROCESSINFO procedure is:

```
{ <error> := } PROCESSINFO ( <cpu,pin>
{ CALL       } ------------ - ---------
                            , <process id>
                            , <creator accessor id>
                            , <process accessor id>
                            , <priority>
                            , <program file name>
                            , <home terminal>
                            , <system number>
                            , <search mode> )
                                            -
```

where

<error>, INT,

indicates the outcome of the call.

```
 0 = status for process <cpu,pin> is returned.
 1 = process <cpu,pin> does not exist.  Status for next
     higher <cpu,pin> is returned.  <process id>[3] =
     <cpu,pin> is process for which status is returned.
 2 = process <cpu,pin> does not exist and no higher
     <cpu,pin> exists.
 3 = unable to communicate with <cpu>.
 4 = <cpu> does not exist.
 5 = the system specified by <system number> could not be
     accessed.
99 = parameter error.
```

<cpu,pin>, INT:value,

specifies the process whose status is being requested.

<process id>, INT:ref:4,

if present, is returned the process ID of the process whose status is actually being returned. Note that this may be different than the process whose status was requested via <cpu,pin> (see <error>)..

→

If multiple search conditions are specified, then all must be met.

example:

```
! return status for all processes run by me at my terminal.

caid := PROCESSACCESSID;
CALL MYTERM ( hometerm );
pin  := 0;
mode := %42000;
WHILE PROCESSINFO ( pin , pid , caid , paid , pri , prog ,
                    hometerm , , mode ) < 2 DO
  BEGIN
    .
    .
    .
    pin := pid [ 3 ] + 1;
  END;
```

CONSIDERATIONS

- If <system number> specifies a remote system, <process id> is returned in network form; otherwise, <process id> is returned in local form.

  If the process's home terminal is in a remote system, then <home terminal> is returned in network form.

  If <system number> specifies a remote system, file names (such as home terminal) are returned in local form (starting with "$").

The SETLOOPTIMER procedure has two uses:

1. To abort the caller if the caller begins looping (i.e., malfunctioning).

2. To permit the caller to calculate the amount of processor time it has used.

A call to the SETLOOPTIMER procedure is used to set the caller's "process loop timer" value. A positive loop timer value enables process loop timing by the operating system and specifies a limit on the total amount of processor time the caller is allowed. If loop timing is enabled, the operating system decrements the loop timer value as the process executes (i.e., is in the active state). If the loop timer is decremented to zero (indicating that the time limit is reached), a "process loop timer timeout" trap occurs (trap no. 4). (Loop timing is disabled by specifying a loop timer value of zero).

The call to the SETLOOPTIMER procedure is:

```
CALL SETLOOPTIMER ( <new time limit> , <old time limit> )
```

where

<new time limit>, INT:value,

specifies the new time limit value, in .01-second units, to be set into the process's loop timer. <new time limit> must be a positive value.

If zero (0) is passed as the <new time limit> value, process loop timing is disabled.

<old time limit>, INT:ref:1,

if present, returns the current setting of the process's loop timer.

Condition code settings:

<	(CCL)	indicates that the <new time limit> parameter was omitted or was specified as a negative value. The state of process loop timing and the setting of the process's loop timer is unchanged.
=	(CCE)	indicates that the <new time limit> value was set into the process's loop timer and that loop timing is enabled.
>	(CCG)	is not returned by SETLOOPTIMER.

→

```
PROC main^proc main;
  BEGIN
      .
      .
    CALL SETLOOPTIMER ( %77777 );
    IF < THEN ...
      .

        enables loop timing.  Time limit value is approximately
        five and one-half minutes.
      .
```

The program executes to completion, then SETLOOPTIMER is called again to obtain the current setting of the loop timer:

```
    CALL SETLOOPTIMER ( 0 , old^val );
    IF < THEN ...
    time^used := %77777 - old^val;

       the total amount of processor time used is calculated by
       subtracting the current setting of the loop timer,
       "old^val", from the original loop timer value (%77777).
    .
  END; ! main^proc.
```

The SETSTOP procedure permits a process to protect itself from being deleted by any process but itself or its creator.

The call to the SETSTOP procedure is:

```
{ <last stop mode> := } SETSTOP ( <stop mode> )
{ CALL               } ------- - ----------- -
```

where

<last stop mode>, INT,

is returned either the preceding value of <stop mode>, or -1 if an illegal mode was specified.

<stop mode>, INT:value,

specifies a new stop mode. The modes are:

0 = stoppable by any process

1 = stoppable only by

- the super ID
- a process whose process accessor ID = this process's creator
- a process whose process accessor ID = this process's accessor ID (this includes the caller to STEPMOM)

2 = unstoppable (privileged users only)

(See "Security System" for an explanation of "super ID" and "process accessor ID").

example:

```
last^mode := SETSTOP( new^mode );
```

CONSIDERATIONS

- The default stop mode when a process is created is 1.
- If a process's stop mode is 1 and a STOP is issued to it by a process without the authority to stop it, the process does not stop; it is deleted, however, if and when the stop mode is changed back to 0.

example:

```
CALL SIGNALTIMEOUT ( 1000D,,, tle );   ! 10 seconds
IF > THEN ...
IF < THEN ...
```

CONSIDERATIONS

- When a TLE set by a call to SIGNALTIMEOUT times out, a system message -22 is read from $RECEIVE. The user buffer then contains a 4-word message, as follows:

<sysmsg>	=	-22
<sysmsg>[1]	=	<parameter1> supplied to SIGNALTIMEOUT (if none supplied, 0)
<sysmsg>[2] FOR 2	=	<parameter2> supplied to SIGNALTIMEOUT (if none supplied, 0D)

  The READ (or AWAITIO) completes with a CCG and error #6.

- This procedure can be used with CANCELTIMEOUT by a multi-threaded i/o process to verify that an i/o operation completes within a certain time. The process calls SIGNALTIMEOUT when initiating the i/o operation, then calls CANCELTIMEOUT after completion if the process has not been signalled on $RECEIVE.

- A process may do exact elapsed timing using the instruction RCLK. This instruction loads registers [D:C:B:A] with a QUAD value of the current time with a resolution of 1 microsecond. For example:

```
FIXED x, y, z;
CODE (RCLK);
STORE x;
  .

CODE (RCLK);
STORE y;
z := y - x;       ! z now contains the elapsed time, in
                  ! microseconds, from the first RCLK to the
                  ! second.
```

- Figure 3-6 illustrates the effect of STEPMOM.

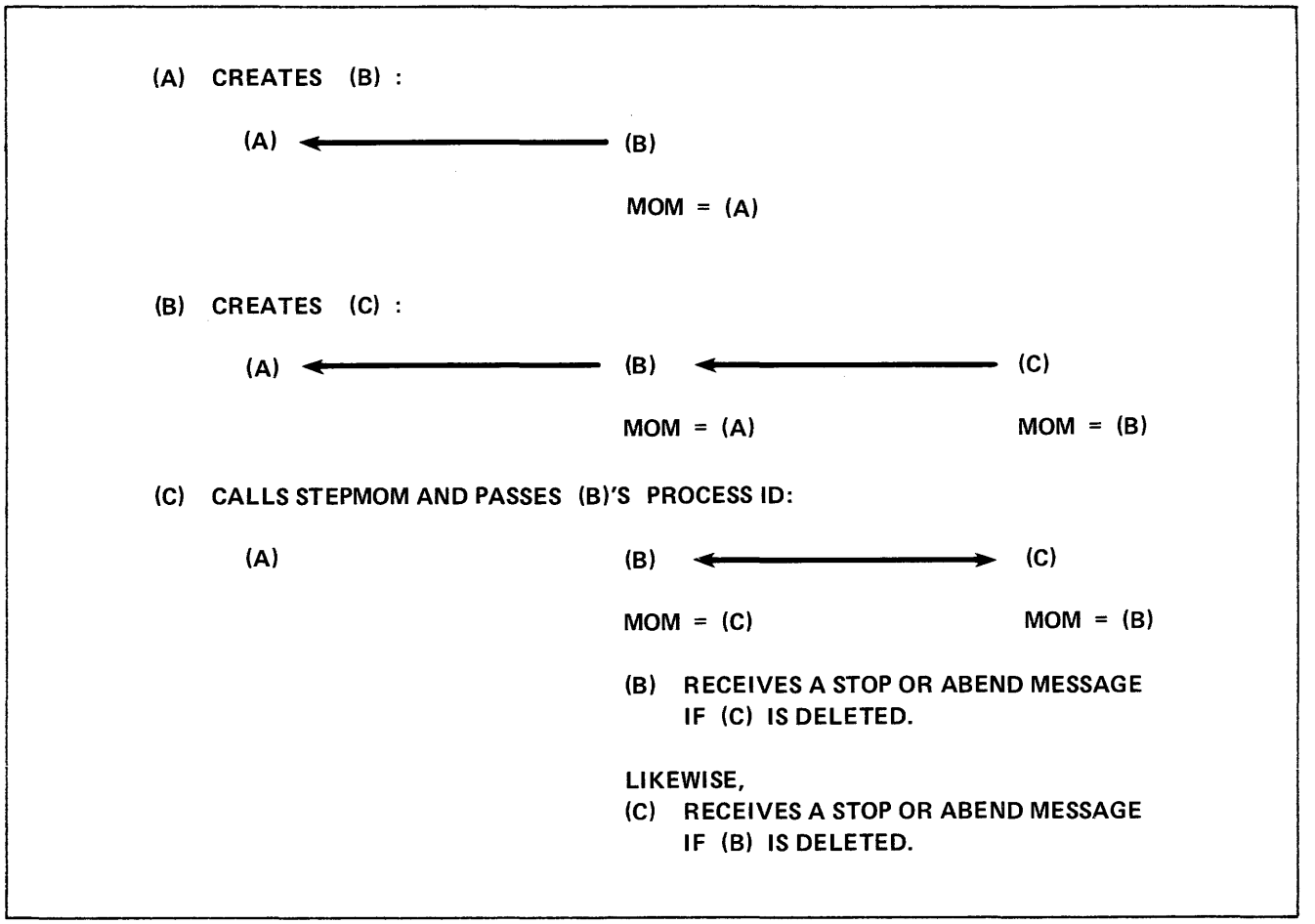


Figure 3-6. Effect of STEPMOM

The SUSPENDPROCESS procedure puts a process [pair] into the suspended state and thereby prevents that process from being active (i.e., executing instructions). (A process is removed from the suspended state and put back into the ready state if it is the object of a call to the ACTIVATEPROCESS procedure.)

The call to the SUSPENDPROCESS procedure is:

```
CALL SUSPENDPROCESS ( <process id> )
---- -------------- - ------------ -

where

   <process id>, INT:ref,

     is an array containing the process ID of the process to be
     suspended.  If <process id>[0:2] references a process pair
     and <process id>[3] is specified as -1, then both members
     of the process pair will be suspended.

condition code settings:

   < (CCL)  indicates that SUSPENDPROCESS failed, or no process
            designated <process id> exists.
   = (CCE)  indicates that <process id> has been suspended.

example:

   CALL SUSPENDPROCESS ( pid );
   IF < THEN ...  ! "pid" doesn't exist.
```

CONSIDERATIONS

- The caller of SUSPENDPROCESS must have the same process accessor ID as the process [pair] it is attempting to activate (see "Security System" for an explanation of "process accessor ID") or be the super ID.

The following example shows the use of the NEWPROCESS procedure to run a program and the use of file management procedures to send and receive a startup message. The example shows the creation of a single non-named process. For an example of how to create a named process pair and the action taken by each member of the pair, see section 5, "Checkpointing Facility".

EXAMPLE

In this example, an application process creates a new process in its own processor module. Following creation of the new process, the creator sends it a startup message (a startup message is the first message sent to a new process):

The following is written in the "creator" application program:

creator

```
  .
INT .pfilename[0:11] := "$VOL1   SVOL3   MYPROG  ",
    .pid[0:11] := 12 * [ "  " ],
     error,
     fnum, .buffer[0:71];
```

NEWPROCESS is called to run "$VOL1 SVOL3 MYPROG " in the same processor module as the creator:

```
  .
CALL NEWPROCESS ( pfilename,,,, pid, error );
IF error.<0:7> > 1 THEN .... ; ! check "error".
```

If the process is created successfully, the new process's process ID is returned in "pid", and zero or one is returned in "error.<0:7>".

Then a file is opened to the new process using the file management OPEN procedure:

```
  .
CALL OPEN ( pid, fnum );
IF < THEN ....; ! open failed.
  .
```

Then a message is sent to the new process:

```
  .
buffer ':=' "GET TO WORK TURKEY";
CALL WRITE ( fnum, buffer, 19 );
  .
```

GENERAL INFORMATION

System processes, such as a process controlling a disc, are subject to the same priority structure as application processes. Therefore, it is important that priorities be assigned in a manner that permit necessary system operations to take place when needed.

For example: Suppose a system process controlling a disc is assigned a priority of 150, and an application process in the same processor module that uses the disc is assigned a priority of 200. Initiating a no-wait operation with the disc does not provide the intended result, because the disc process, having a lower priority, never gets a chance to execute. Only when the application process is suspended, because of a call to the AWAITIO procedure, does the disc process finally execute and complete the i/o operation.

SUGGESTED PRIORITY VALUES

The following are recommended priority values for user processes:

system processes	priority	user processes
disc i/o processes	220	
	}	only processes that do not use virtual memory (i.e, resident)
Memory Manager	210	
Operator	210	
System Monitor	200	
non-disc i/o processes	200	
	}	command interpreters used to run application processes
	}	application processes
	150	command interpreters used for program development
	149	editors used for program development (this priority is assigned automatically by command interpreters running at priority 150)
	}	
	145	spoolers used for program development
	}	
	140	compilers and background batch processing
	}	
	1	?

```
  :
INT       .prog 1 [0:11]  := "$SYSTEM SYSTEM SORT     ";
          .prog 2 [0:11]  := "$VOL1    SVOL    LOG    ";
LITERAL pri  200=200, pri  150=150,cpui=1;
INT pid1,pid2,pid3,error;
  :
CALL NEWPROCESS (prog  1,pri  200,3 cpu1,pid1,error);
  :  (1,7 returned in <pid1>)
CALL NEWPROCESS (prog  1, pri  200,2, cpu1,pid2,error);
  :  (1,8) returned in <pid2>)
CALL NEWPROCESS (prog  2,pri  150, 4, cpu1,pid3,error);
  :  (1,9) returned in <pid3>)
```

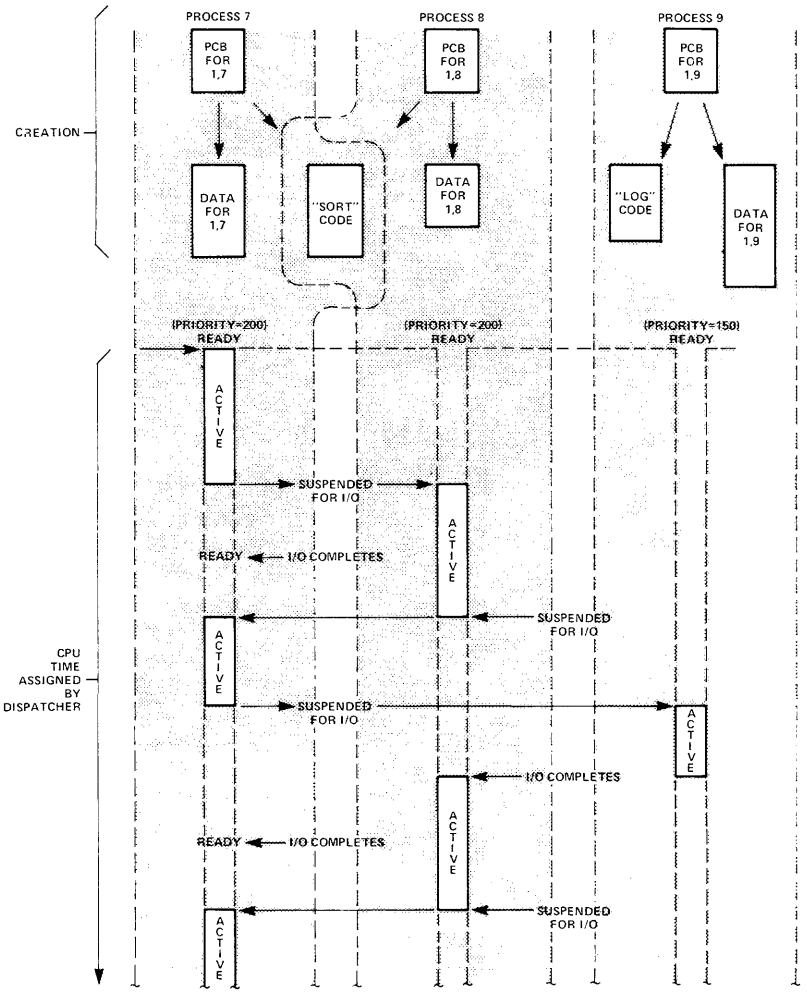


Figure 3-7. Execution Priority Example

INDEX

READER'S COMMENTS

Tandem welcomes your feedback on the quality and usefulness of its publications. Please indicate a specific *section* and *page* number when commenting on any manual. Does this manual have the desired completeness and flow of organization? Are the examples clear and useful? Is it easily understood? Does it have obvious errors? Are helpful additions needed?

Title of manual(s): _____

FOLD ➤

FOLD ➤

FROM:

Name _____

Company _____

Address _____

City/State _____ Zip _____

A written response is requested, yes no ?

82336 A00

TANDEM COMPUTERS INCORPORATED
19333 Vallco Parkway
Cupertino, CA 95014