NonStop™ Systems

GUARDIAN™ Operating System Programmer's Guide

Operating System Library

82357

NOTICE

Effective with the B00/E08 software release, Tandem introduced a more formal nomenclature for its software and systems.

The term "NonStop 1+™ system" refers to the combination of NonStop 1+ processors with all software that runs on them.

The term "NonStop™ systems" refers to the combination of NonStop II™ processors, NonStop TXP™ processors, or a mixture of the two, with all software that runs on them.

Some software manuals pertain to the NonStop 1+ system only, others pertain to the NonStop systems only, and still others pertain both to the NonStop 1+ system and to the NonStop systems.

The cover and title page of each manual clearly indicate the system (or systems) to which the contents of the manual pertain.

NonStop™ Systems

GUARDIAN™ Operating System Programmer's Guide

Abstract
This manual describes the interface between user programs and the GUARDIAN Operating System on the NonStop systems.

Product Version
GUARDIAN B00

Operating System Version
GUARDIAN B00 (NonStop Systems)

Part No. 82357 A00

March 1985

Tandem Computers Incorporated
19333 Vallco Parkway
Cupertino, CA 95014-2599

DOCUMENT HISTORY

Edition	Part Number	Operating System	Date
First Edition	82357 A00	GUARDIAN B00	March 1985

NEW AND CHANGED INFORMATION

This is a new publication for the NonStop system. The existing two-volume GUARDIAN Operating System Programming Manual (part numbers 82336/82337 with updates 82189/82192) is replaced by two separate manuals for the B00 release of the GUARDIAN operating system: this GUARDIAN Operating System Programmer's Guide and the System Procedure Calls Reference Manual (part number 82359).

All of the procedure call syntax information is now contained in the System Procedure Calls Reference Manual.

Note that the scope of the System Procedure Calls Reference Manual has been enlarged beyond that of the existing GUARDIAN Operating System Programming Manual to include other products. The scope of the GUARDIAN Operating System Programmer's Guide has not been enlarged. It explains only how to use those features in the existing GUARDIAN Operating System Programming Manual. "How-to" information on procedure calls that are part of other products, such as the spooler, ENFORM, and SORT/MERGE, continues to reside in the manuals for those products.

For the B00 release, the following new features were added to this manual:

- The time services provided for the GUARDIAN operating system have been redesigned. The main features include:

  --Four-word, microsecond resolution
  --Julian Date based (GMT) timestamps
  --CPU clock rate averaging
  --Clock rate adjustment
  --Automatic daylight savings time adjustments
  --Julian date conversion routines
  --Callable procedure to set system clocks

  These features are described in Section 16.

- Software support for the new 5530 serial printer is described in Section 7.
- New $CMON functions are described in Section 5.
- LOCKMEMORY and UNLOCKMEMORY were removed because they are privileged procedures.
- Operator console information in Section 10 has been updated to reflect the latest changes.
- The Trap Handling description in Section 13 has been rewritten.
- A number of minor corrections and clarifications have been made to the manual.
- The entire publication has been reorganized and reformatted.

CONTENTS

FIGURES

TABLES

PREFACE

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

Specifically, this manual discusses:

- Managing files, managing processes, and checkpointing data using the procedures provided by the GUARDIAN operating system
- Interfacing between application programs and the GUARDIAN command interpreter (COMINT)
- Performing input/output using the sequential I/O procedures and the INITIALIZER procedure
- Formatting input and output using the formatter
- Using a trap handler
- Managing extended data segments

AUDIENCE

This manual is for systems and application programmers with special needs to call operating system procedures from their programs. The audience level is assumed to be intermediate- to advanced-level programmers not familiar with the GUARDIAN operating system. Familiarity with the Tandem Transaction Application Language (TAL) or some other programming language, such as FORTRAN or COBOL, is required.

SUGGESTED READING

Prerequisite reading includes:

- Introduction to Tandem Computer Systems (Part No. 82503) for a general overview of the system
- GUARDIAN Operating System User's Guide (Part No. 82396), Sections 1, 2, and 3, for information about logging on to the system and running programs in general

Required reference manuals are:

- System Procedure Calls Reference Manual (Part No. 82359) for all procedure call syntax and considerations
- GUARDIAN Operating System Utilities Reference Manual (Part No. 82403) for information not covered in the above manuals

For more information regarding the Tandem NonStop systems, refer to the manuals listed below.

- System Description Manual (Part No. 82507)
- System Operator's Guide (Part No. 82401)
- System Management Manual (Part No. 82569)
- System Messages Manual (Part No. 82409)
- Transaction Application Language (TAL) Reference Manual (Part No. 82581)
- Transaction Monitoring Facility (TMF) Reference Manual (Part No. 82341)
- Transaction Monitoring Facility (TMF) System Management and Operations Guide (Part No. 82543)
- ENSCRIBE Programming Manual (Part No. 82583)
- EXPAND Reference Manual (Part No. 82370)
- EXPAND Network Design Guide (Part No. 82371)

- ENVOY Byte-Oriented Protocols Reference Manual (Part No. 82582)
- ENVOYACP Bit-Oriented Protocols Reference Manual (Part No. 82588)
- SORT/MERGE User's Guide (Part No. 82091)
- Spooler Programmer's Guide (Part No. 82394)
- BINDER Manual (Part No. 82514)
- CROSSREF Manual (Part No. 82516)
- INSPECT Interactive Symbolic Debugger User's Guide (Part No. 82315)
- DEBUG Manual (Part No. 82598)
- Communications Utility Program (CUP) Reference Manual (Part No. 82430)
- X.25 Access Method--(X25AM) (Part No. 82431)
- Device-Specific Access Methods--(AM3270/TR3271) (Part No. 82432)
- Device-Specific Access Method--(AM6520) (Part No. 82433)

You will want to refer to other reference and programming manuals, especially for communications products you are using. For a complete list of Tandem software technical manuals and their part numbers, refer to the following publication:

- Guide to Software Manuals (82552)

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

- Master Index (82586)

SYNTAX CONVENTIONS IN THIS MANUAL

The following list summarizes the conventions for syntax notation in this manual.

Notation	Meaning
UPPERCASE LETTERS	Uppercase letters represent keywords and reserved words; you must enter these items exactly as shown.
<lowercase letters>	Lowercase letters within angle brackets represent variables that you must supply.
Brackets []	Brackets enclose optional syntax items. A vertically aligned group of items enclosed in brackets represents a list of selections from which you may choose one or none.
Braces {}	Braces enclose required syntax items. A vertically aligned group of items enclosed in braces represents a list of selections from which you must choose only one.
Ellipsis ...	An ellipsis immediately following a pair of brackets or braces indicates that you can repeat the enclosed syntax items any number of times.
Percent Sign %	Precedes a number in octal notation.

SECTION 1

INTRODUCTION TO THE GUARDIAN OPERATING SYSTEM

The basic design philosophy of the Tandem NonStop system is that no single module failure will stop or contaminate the system. This capability is called fault-tolerant operation.

Redundant hardware, backup power supplies, alternate data paths and bus paths, redundant controllers, and mirrored discs all contribute to the fault-tolerance of the NonStop system. The Introduction to Tandem Computer Systems describes these features.

There is more to a fault-tolerant system than just hardware. Fault tolerance requires that all programs, operating system as well as individual application programs, contribute to the reliability and recoverability of a process in the case of a failure. Therefore, fault tolerance should be considered from both the hardware and the software perspectives.

Fault-tolerant software at the application level is achieved by the use of process pairs; a primary process performs the application, while a secondary (backup) process in another CPU remains ready to take over if the primary fails. If the primary fails, the backup process resumes work at the point of the last valid checkpoint. The use of checkpoints is explained in Section 12.

One of the most effective safeguards against loss of data is the use of mirrored disc volumes. Mirrored volumes allow you to maintain copies of data on two physically independent disc drives that are accessed as a single device and managed by the same I/O process. All data written to one disc is written to the other as well. All data read from one disc could be read as well from the other because the data is identical. A mirrored volume safeguards your data against single disc failures; if one disc drive fails, the other should still be operational. The odds against both disc drives of a mirrored pair failing at the same time are quite great.

After a disc is replaced or a drive is repaired, all data is copied back onto it while transaction processing continues. Mirrored-pair operation resumes as this transfer of data occurs.

The GUARDIAN operating system provides both multiprocessing (parallel processing in separate processor modules) and multiprogramming (interleaved processing in one processor module).

In a typical NonStop system, master copies of the GUARDIAN operating system, configured for the specific application, are kept in a "system" area. Critical and frequently used parts of the GUARDIAN operating system are always present in each processor module memory. As such, the system capabilities are maintained even if a processor module, I/O channel, or disc drive fails. Noncritical or less frequently used parts of the GUARDIAN operating system are brought into a processor module's memory from disc only when needed.

Maintenance of the system area and operation of mirrored volumes is entirely transparent to both application programs and system users. Several other functions of the GUARDIAN operating system are also transparent to application programs. These include:

- Scheduling processor module time among multiple processes according to their application-assigned priorities (a process is an executing program)
- Enabling 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
- Preparing a program for execution in virtual memory when a request is made to run a program.

The GUARDIAN operating system provides an extremely important additional function. Concurrent with application program execution, the operating system continually checks the integrity of the system. Each processor module transmits "I'm alive" messages to every other processor module at a predefined interval (typically once per second). Following this transmission, 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 the "I'm alive" 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 nontransmitting processor module is inoperative; if it cannot, 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.

In addition to the safeguards offered by the GUARDIAN operating system, application programs must also contribute to their own fault tolerance. Each application program must ensure that it has a backup process.

An application program "sees" operating system services as a set of callable library procedures. The library procedures have names such as READ, WRITE, OPEN, and so on. For example, to request the operating system service for input, a call to the operating system READ procedure is written in the application program. (The operating system library procedures exist in the system code area and therefore are shared by all processes.)

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

- File system--how to perform the input and output operations is discussed in Section 2.
- Process control--how to run, suspend, and stop programs is described in Section 3.
- System messages--how to communicate information from the GUARDIAN operating system to application processes is described in Section 4.
- Command interpreter program--how to communicate run-time information to an application process is described in Section 5.
- Terminals--how they relate to the operating system, as well as an overview of terminology, access, connection, and error recovery is presented for the TERMPROCESS interface in Section 6.
- Line printers--the interfaces between line printers and the operating system, along with the use of SETMODE and CONTROL operations in setting up and using printers, is discussed in Section 7.
- Magnetic tape--the characteristics and procedures used to control tape usage are presented. Concepts, programming considerations, and CONTROL operations are described. BCD/ASCII character sets and available conversion modes are also presented in Section 8.
- Card readers--applicable procedures are described in Section 9.

- Operator console--its use as a logging device for system errors, statistical information, or application-supplied information is described in Section 10.
- Fault-tolerant programming--use of the Transaction Monitoring Facility (TMF) and its interface with the GUARDIAN operating system is discussed in Section 11.
- Checkpointing facility--for writing fault-tolerant (NonStop) programs is described in Section 12.
- Traps and trap handling--how to programmatically handle trapped programs and identify critical error conditions is described in Section 13.
- Extended memory management--how to allocate and use extended memory segments and pools is described in Section 14.
- Memory management techniques--how to influence the efficiency of a process is presented in Section 15.
- Utility procedures--procedures for time functions, timestamp translation, string and number manipulation, number translation, and other services, such as the INITIALIZER, are described in Section 16.
- Sequential I/O--an alternate, standardized set of procedures for performing common input-output operations only for sequential files is described in Section 17.
- Formatting--the formatting of output data and conversion of input data is discussed in Section 18.

THE FILE SYSTEM

A file (Figure 1-1) is the symbolic representation of:

- an input-output device,
- a process, or
- the operator console,

for purposes of performing input-output operations in a simple, efficient, and uniform manner. Typical file names are shown.

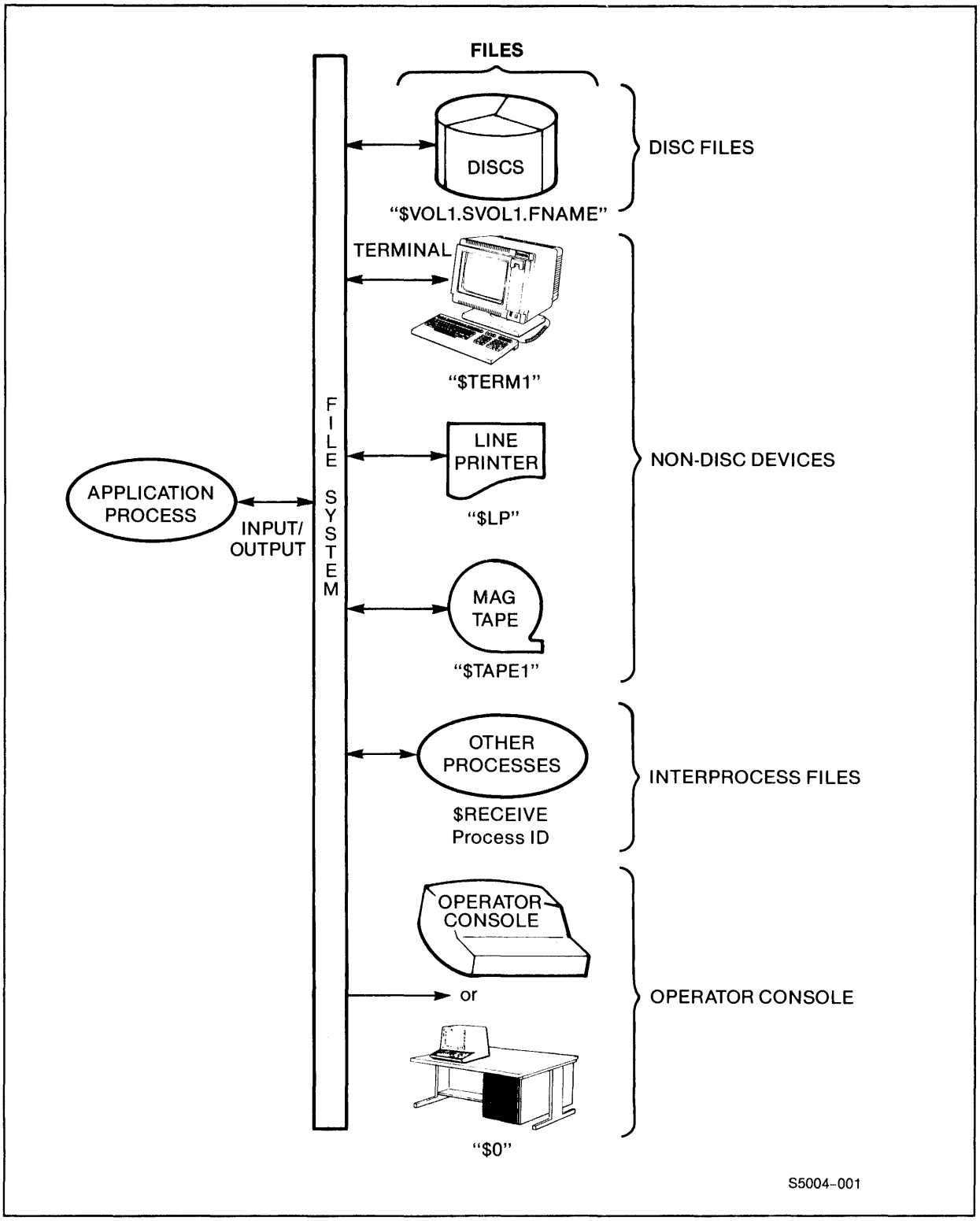


Figure 1-1. Files

Each file, device, process (and even the console) in the system is identified by a unique file name. Devices that are normally dedicated to a single process or a related set of processes while in use, such as terminals or line printers, are represented by a single file name. In the case of disc devices, because they are capable of storing massive amounts of data and must be accessed by several processes concurrently, a file name represents a portion of the total storage area on a designated disc. In short, all nondisc devices, all disc files, and all processes are treated as files.

PROCEDURES

There is a unique procedure defined for each operating system operation. Each procedure has a name such as READ, WRITE, OPEN, CLOSE, and so on.

Because there are so many GUARDIAN operating system procedures, they are divided into groups by function for presentation in this guide. Each group has a descriptive name. For example, those procedures that perform file operations are called the file system procedures. The procedures that control running programs, or processes, are called process control procedures. The procedures provided to perform utility operations are called utility procedures.

All GUARDIAN procedure syntax is presented alphabetically in the System Procedure Calls Reference Manual.

GUARDIAN operating system functions can be accessed in two ways: through the command interface provided by the GUARDIAN command interpreter (see the GUARDIAN Operating System User's Guide), or through the programmatic interface described in this guide.

An alternate set of procedures, named the Sequential Input/Output Procedures (or SIO procedures), is also available for use under certain conditions. See Section 17.

File operations are performed by making calls to GUARDIAN file system procedures. All files are accessed through this same set of procedures, thereby providing a single, uniform access method. Additionally, the file-system procedures are designed to eliminate the operating peculiarities of various devices. The file-system procedures include:

CREATE	defines a new file on a disc volume
OPEN	provides access to a file

READ	transfers data from a file to an application process data area
WRITE	transfers data to a file from an application process data area
WRITEREAD	writes data to a file, then waits for data to be returned (read) from the file
READUPDATE	reads data from a file in anticipation of updating a record in the case of disc files or replying to a request message in the case of processes (the disc update is made by using WRITEUPDATE; the message reply is made by using REPLY)
REPLY	is used to send a reply message in response to reading a request message by using READUPDATE
WRITEUPDATE	writes an updated record to a disc
CLOSE	terminates access to a file
PURGE	deletes a disc file

In order to access a file, it must first be opened. This is done by using a call to the file-system OPEN procedure:

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

<Filename> is an array in the program data area containing the symbolic name of the file. <Filenum> is a value returned by OPEN to identify the file in subsequent file-system calls.

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

```
CALL WRITE(filenum,buffer,write-count);
```

<Buffer> is an array in the program's data area containing the information to be written. <Write-count> is the number of bytes to be written.

To read (input) from the same file, use:

```
CALL READ(filenum,buffer,read-count,count-read);
```

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

PROCESSES

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 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 more complete information, refer to Section 3):

- Process creation

  The term "process creation" refers to the action performed by a special system process called the System Monitor which initially prepares a program for execution. Process creation can be initiated by application programs or by the GUARDIAN command interpreter (COMINT) through the process control NEWPROCESS and NEWPROCESSNOWAIT procedures.

  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, input and output file names, and any application-dependent parameters specified through the RUN command. The startup message can be read by the new process through use of the standard GUARDIAN file-system procedures, or it can be obtained by using the INITIALIZER procedure (see "Communicating With Other Processes" in Section 4).

- 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.

--A creator has the right to delete (stop) processes it has created.

- Process deletion

  Process deletion is the act, by the operating system, of stopping further process execution.

  There are two types of process deletion: normal and abnormal. Normal deletion is initiated by a call to the process control STOP procedure. Abnormal deletion is initiated by a call to the process control ABEND procedure or by the occurrence of a trap when certain other conditions are present (see "Traps" in Section 13).

  Process deletion can be initiated by a process itself, by another process (under some circustances) or, if an abnormal deletion, by the operating system.

- Process ID

  A process is uniquely identified throughout the system by its process ID. There are two forms of the process ID. The first is the timestamp form. This form of process ID is assigned to a process at process creation time by the operating system and contains a timestamp of when the process was created. The other form of process ID is the process name form, which is commonly used when a pair of processes must be identified by a common name. Process names are application-defined and are assigned to processes at process creation time. Either form of process ID can be used to identify processes for the purpose of interprocess communication. (Interprocess communication is the sending of messages between processes. It is performed through use of the file system.) The use of the process name, moreover, has the advantage that a process can be known throughout the system by a predefined identifier.

- Destination Control Table (DCT)

  Named processes are known throughout the system by means of entries in the destination control table. Parts of this table were known in earlier Tandem systems as the process-pair directory (PPD). (A device table was combined with the PPD of the NonStop 1+ system to form the DCT in NonStop systems.) The DCT is the table that contains the names of all named processes in the system. PPD is the name used in this manual

to describe just the named processes in the DCT. A process name is entered into the DCT when a process having a name is created. A given process name is deleted from the DCT when the last process having that name is deleted. (See "Process Pair".)

- Home terminal

  Associated with each process is a home terminal. The home terminal is the terminal from which the command interpreter RUN command is given to run the process. This terminal is used by the system to communicate with the programmer during the debugging phase of program development (see "DEBUG Facility") and to display process-related error messages. Additionally, the home terminal can be used by the application program to communicate interactively with the person who runs the program.

Process Structure

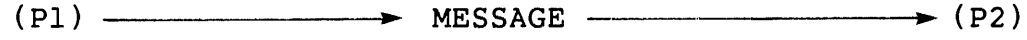
The process structure provided by the GUARDIAN operating system allows a program to be written as though it could run on a processor of its own. This abstraction is possible because:

- Each process executes independently of and without interference from all other processes
- Each process 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 should execute correctly 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 between one another using messages.

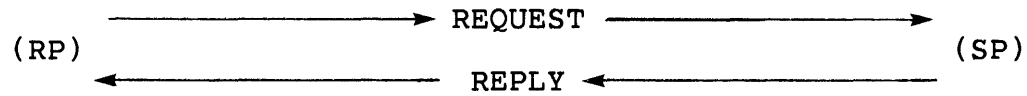


(P) = process

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:

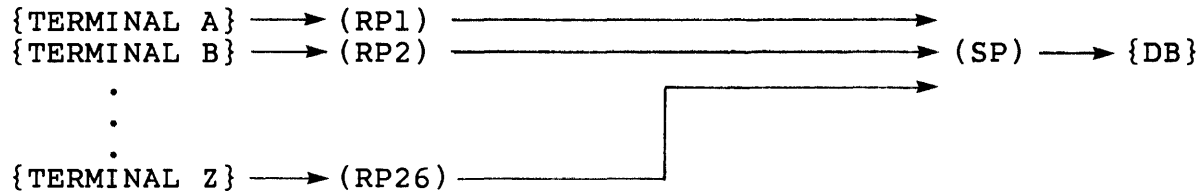
Applications are structured in much the same way as the operating system. That is, specific functions are performed by independent processes that communicate with each other by way of interprocess messages.

A common structure for applications is the requester-server relationship. With this structure, requester processes make requests of a common server process (an application may consist of several of these requester-server relationships). A request is made in the form of an interprocess message (sent through the file system). The server replies to the message through the file system (the reply usually consists of the requested data).



(RP) = requester process (SP) = server process

For example, in a simple data base query application, each user terminal is controlled by a separate requester process. The function of the requester process is to accept and interpret commands entered at the terminal, then send a request for a data base record to the common server process. The function of the server process is to accept a request from a requester, then return the requested record to that requester.



The obvious benefits of this structure are:

- The application is modularized by function: terminal device control and data base control.
- The requester program is written to control a single terminal. (To control multiple terminals, the program is run multiple times concurrently; each time, a different terminal is specified as the device to be controlled).

The requester-server process relationship is discussed in detail in Section 4, "Communicating With Other Processes" and examples are presented in Appendix B.

Process Pairs

It is possible for a properly coded application process to recover from any type of hardware failure except one--a failure of the processor module where it is executing. Because of this, much use is made of NonStop process pairs. This is a method of increasing fault tolerance. For each pair of processes, the primary process executes in one processor module, and the backup process executes in another.

A process pair is usually two executions of the same program. Logic in the program determines whether the process is executing in the primary mode to perform the designated work, or in the backup mode to monitor the operability of the primary.

With this primary and backup structure, the backup process is continually aware of the executing state of the primary process by the use of checkpointing messages periodically sent from the primary process to the backup process. When the backup is informed its primary has failed (by the receipt of a process or processor failure system message), the backup switches into the primary mode and continues with the application's work.

A process pair is typically identified by a single process name. A process pair's process name is entered into the PPD part of the DCT when the first process of the pair is created. At this time, the identity of the ancestor process is also entered into the DCT. (An ancestor process is the process responsible for creating the first member of a process pair). The DCT provides capabilities that are useful for fault-tolerant 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 DCT (the latter occurs when the last process associated with a process name stops or fails). Other important fault-tolerant considerations when communicating with named process pairs are described in Section 2.

Process Control Functions

Process control operations are performed by calling the GUARDIAN process control procedures. These procedures are described in Section 3. Examples of process control procedures include:

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

For example, to create a process (run a program), the process control NEWPROCESS procedure might be called as follows:

```
CALL NEWPROCESS (progname,pri,mem,cpu,process^id,error,name);
```

<progname> is an array in the calling program data area containing the file name of the program to be run. <pri> is the execution priority to be assigned to the new process. <mem> is the number of data pages to be allowed for the new process, and <cpu> is the processor module number where the new process is to be created. <process^id> is a value returned by NEWPROCESS to identify the newly created process, and <error> is a value returned to indicate whether or not the process creation was successful. <name> is the process name to be assigned to the new process. The creator process can then pass the <process^id> to the file-system OPEN procedure as a file name, and then use calls to WRITE or WRITEREAD to send startup information to the new process.

Or to have a process delete itself, the STOP procedure is called:

```
CALL STOP;
```

The calling process is deleted, all the files it had opened are closed and its creator receives a STOP system message.

To suspend a process for some designated period of time, the DELAY procedure can be called in the following manner:

```
CALL DELAY(1000D);
```

This delays the caller for ten seconds. ("D" indicates that the number represents a 32-bit integer value.)

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 that can be sent to processes are:

```
CPU Down --Processor module failed.
STOP     --Process stopped executing.
ABEND    --Process stopped executing because of abnormal
           condition.
CPU Up   --Processor module reloaded.
```

The system messages are presented in the System Messages Manual.

CHECKPOINTING FACILITY (FAULT-TOLERANT PROGRAMMING)

The checkpointing facility provides the capability for writing application programs that can recover from a processor module failure. To use the checkpointing facility, an application program must be executing as a process pair.

As shown in Figure 1-2, the checkpointing facility is used by the primary process of a process pair to checkpoint pertinent data to its backup process. It is used by the backup process to receive the checkpoint data and to monitor the status of the primary process. (The checkpoint data is sent from the primary process to its backup process in the form of an interprocess message.) If the backup process is notified of the failure of its primary process, the checkpointing facility causes the backup process to begin executing at the point indicated by the latest checkpoint message. (The notification to the backup that the primary has failed is in the form of a CPU Down, STOP, or ABEND system message.)

You must use the following two procedures to checkpoint a process environment. Their use is explained in Section 12; their syntax is given in the System Procedure Calls Reference Manual.

1. CHECKPOINT is called by a primary process to checkpoint its current state to its backup process.

2. CHECKMONITOR is called by a backup process to monitor its primary and take appropriate action in the event of a failure of the primary process.

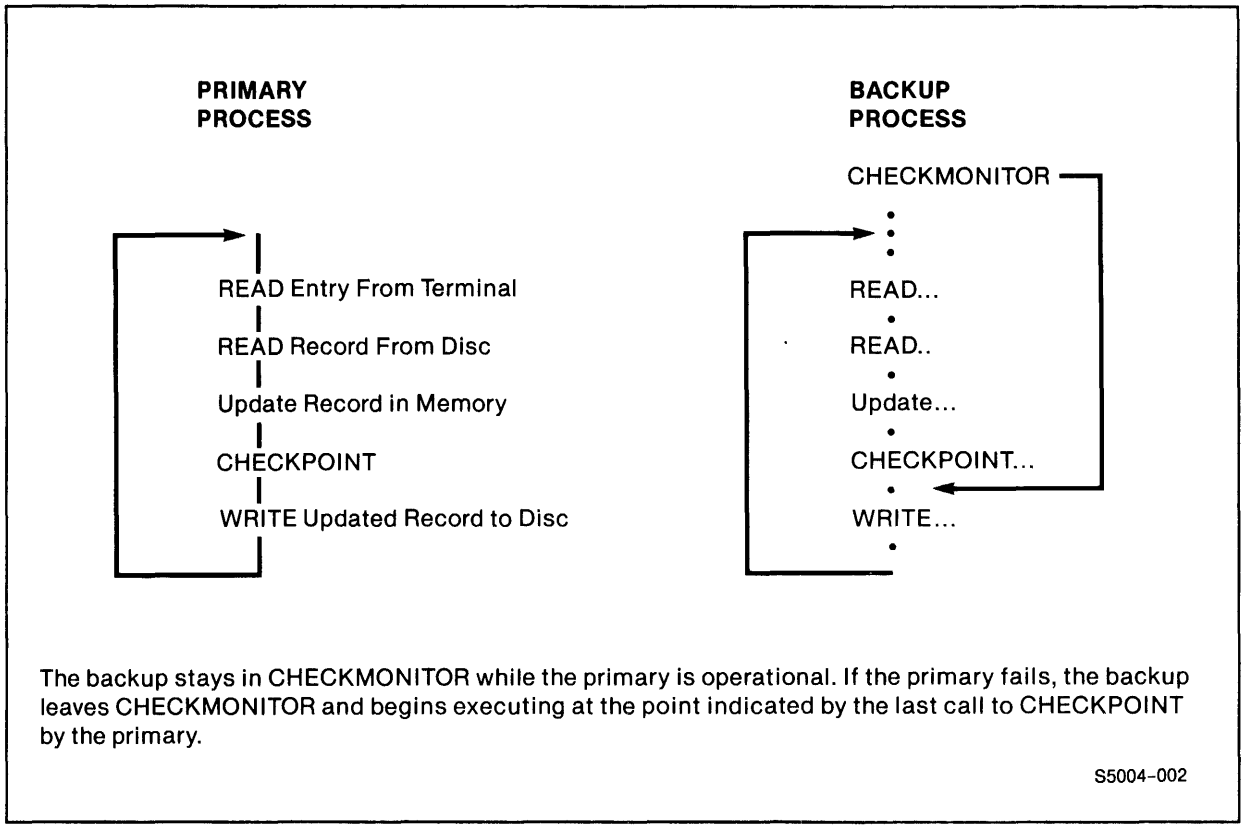


Figure 1-2. Checkpointing

When the checkpointing facility is used, each process in a process pair has the same set of files open, as shown in Figure 1-3. This ensures that the backup process has immediate access to the files in the event of the primary's failure.

Use the following two procedures when opening or closing files in a fault-tolerant environment:

1. CHECKOPEN is called by a primary process to open a file in its backup process.

2. CHECKCLOSE is called by a primary process to close a file in its backup process.

A complete fault-tolerant programming example is presented in Appendix B.

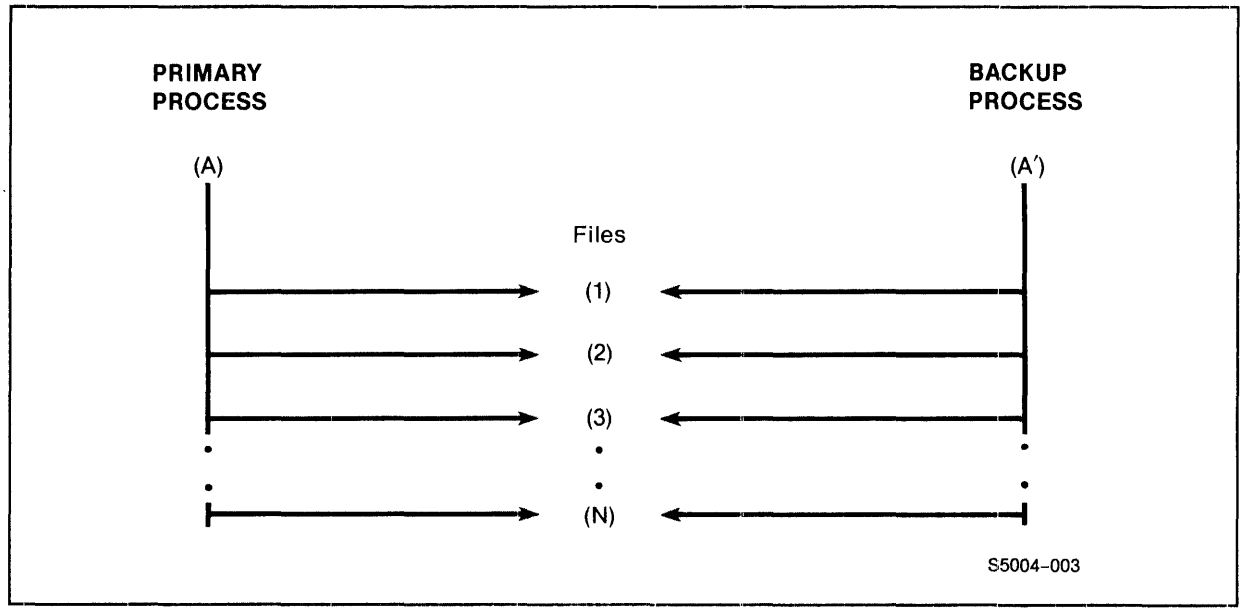


Figure 1-3. Files Open by a Primary/Backup Process Pair

TRANSACTION MONITORING FACILITY (TMF)

If the Transaction Monitoring Facility (TMF) is available on your system, the need to checkpoint programs is practically removed. Checkpointing is usually not necessary when TMF is used because TMF does essentially everything that can be achieved with the NonStop process pairs typically used when checkpointing.

TMF uses audit trails, online dumps, and backout, rollforward, and autorollback facilities to ensure data base consistency even through a total system failure. If a transaction is aborted, all effects of that transaction are removed from the data base. If a system should fail, TMF can remove all effects of any transaction that was interrupted. The other major service of TMF is transaction concurrency control.

The GUARDIAN interface to TMF is described in Section 11.

UTILITY PROCEDURES

Among the various utility procedures are those that pertain to time, system clocks, real time, process time, and so on. Procedures are available to automatically convert Julian day numbers to the Gregorian calendar date and time of day. Time of day at different locations can be based on Greenwich Mean Time, local standard time or, in the United States, on daylight saving time. Procedures are available for setting and converting dates and time of day.

Other utility procedures are presented together because they share an ability to manipulate strings, arrays, or numbers. Among these are procedures used to edit strings of characters, sort arrays of equal-size elements in place, or convert the ASCII representation of a number into its binary equivalent.

The remaining utility procedures include those that do not fit into categories. Among these procedures are included very important procedures such as the INITIALIZER that can reduce the amount of code you need to write to start a program, and make the coding much easier and more consistent.

Use of these utility procedures is described in Section 16; their syntax is given in the System Procedure Calls Reference Manual.

USING THE COMMAND INTERPRETER

The GUARDIAN command interpreter (COMINT) is an interactive program used to run programs, check system status, create and delete disc files, and alter system hardware states. An important feature of the command interpreter is its ability to pass user-specified parameter information to a process at run time. The parameter information is delivered to the process in the form of interprocess messages. The programmatic interface to the GUARDIAN command interpreter is covered in Section 5. The user interface is described in the GUARDIAN Operating System User's Guide. Complete syntax of all COMINT commands is given in the GUARDIAN Operating System Utilities Reference Manual.

EXTERNAL DECLARATIONS FOR OPERATING SYSTEM PROCEDURES

Like all other procedures in an application program, the operating system library procedures must be declared before they can be called. These procedures are declared as "external" to the application program. Declarations for these procedures are provided in a system file designated "$SYSTEM.SYSTEM.EXTDECS0". You should include a SOURCE compiler command specifying this file in the source program following the global declarations but preceding the first call to one of these procedures:

```
<global-declarations>

?SOURCE $SYSTEM.SYSTEM.EXTDECSn ( <ext-proc-name> , ... )

<procedure-declarations>
```

Each external procedure that is referenced in the program should be specified in the SOURCE command.

For example:

```
?SOURCE $SYSTEM.SYSTEM.EXTDECS0 ( 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 for the current release.

Multiple versions of EXTDECS are available. To specify the current version, use EXTDECS0. To specify other versions, use one of the following:

```
EXTDECS0  --current operating system release version
EXTDECS1  --current release, minus 1
EXTDECS2  --current release, minus 2
EXTDECS   --current release, minus 2
```

Because the version is specified by a relative value, there is no need to constantly update the level of EXTDECS used in your programs. If you specify the level as EXTDECS0, your programs will always use the current release level of EXTDECS.

SECURING YOUR FILES

The GUARDIAN operating system 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 who are to access the system.

For each file, file access at each level may be restricted to reading, writing, executing, 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 who is designated the group manager; the group manager is responsible for creating new users in its group. The super ID may also create new users in any group and has full access to any file in the system.

Additional system security is provided by licensing. Programs that are to execute in privileged mode must be licensed by the super ID or be run by the super ID. An attempt by a user other than the super ID to run an unlicensed privileged program is rejected. Processes can be stopped or debugged only by their creator or by the super ID. Privileged processes can be debugged only by the super ID. For more information regarding security, see the GUARDIAN Operating System User's Guide.

TRAPS AND TRAP HANDLING

Certain critical error conditions occurring during process execution prevent the normal execution of a process. These errors, which are for the most part unrecoverable, cause traps to operating system trap handlers. The conditions are:

- Illegal address reference
- Instruction failure
- Arithmetic overflow
- Stack overflow
- Process-loop-timer timeout
- Memory manager disc read error
- No memory available
- Uncorrectable memory error

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

The default trap handler is DEBUG (or INSPECT, if specified for the process). If you do not specify a trap handler, the default is used.

If you prefer to write the code to handle your own traps, you can call the system procedure ARMTRAP. Your trap handler is notified of the particular trap condition. ARMTRAP is described further in Section 13.

DEBUG FACILITY

The GUARDIAN debug facility provides a tool for interactively debugging a running process at the process's home terminal. DEBUG is the default trap handler for all processes as described in Section 13. For a description of the debug facility and instructions for using it, see the DEBUG Reference Manual for your system.

INSPECT

INSPECT is an interactive symbolic debugging tool used to isolate errors in programs. It offers two modes of operation: low-level INSPECT and high-level INSPECT. Low-level operation is very similar to DEBUG. High-level mode INSPECT is much more informative, but it also requires much more memory and table space than DEBUG. Because high-level INSPECT allows the use of symbols instead of address expressions, it is easier to use than DEBUG. Refer to the INSPECT Interactive Symbolic Debugger User's Guide for more information.

If INSPECT is available on your system, it can be specified for use in debugging. If INSPECT is specified for use but is not available, DEBUG is used.

BINDER

BINDER is yet another development tool. If you wish to use high-level INSPECT for a program, the program must have been created with BINDER, or with a compiler that interfaces with BINDER. BINDER builds the tables that go into the object file used by INSPECT. If the symbol table is not appended to the object code, INSPECT can still be run but will have to be used in low-level mode. See the BINDER User's Manual for a complete description of BINDER. There is no relationship between DEBUG and BINDER.

DIVER AND DELAY

The DIVER and DELAY programs are used to facilitate testing of user application programs that are run as NonStop process pairs. DIVER causes a processor to fail and then makes the processor ready for a reload. It is typically used in conjunction with the command interpreter and the DELAY program to automatically cause repeated failures and reloads of processors in a system for test purposes. Fault-tolerant application processes should continue running even while processors are halted and reloaded.

The DIVER and DELAY programs are now documented in the GUARDIAN Operating System Utilities Reference Manual.

SECTION 2

BASIC CONCEPTS: FILES AND FILE NAMES

In a Tandem system, most entities are treated as files. Each disc file, nondisc device, process, and even the operator console is identified by a unique file name.

FILES

Input-output operations are performed by transmitting blocks of data between processes and files. A file can be all or a portion of a disc, or a device such as a terminal or line printer, or a process (any running program), or the operator console. A file is referenced by the symbolic file name that is assigned when the file is created.

Disc Files

The ENSCRIBE data base record manager, an integral part of the GUARDIAN operating system, provides access to and operations on disc files. The ENSCRIBE software supports 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.

- Unstructured files--Records are defined by the application process; records are written to and read from a file on the basis of relative byte addresses within the file.

For more information on the ENSCRIBE data base record manager, refer to the ENSCRIBE Programming Manual.

The symbolic name that identifies an individual disc file in the system consists of three parts as shown in Figure 2-1: (1) a volume name to identify a particular disc pack in the system, (2) a subvolume name to identify the disc file as a member of a related set of files on the volume (as defined by the application), and (3) a disc file name to identify the file within the subvolume.

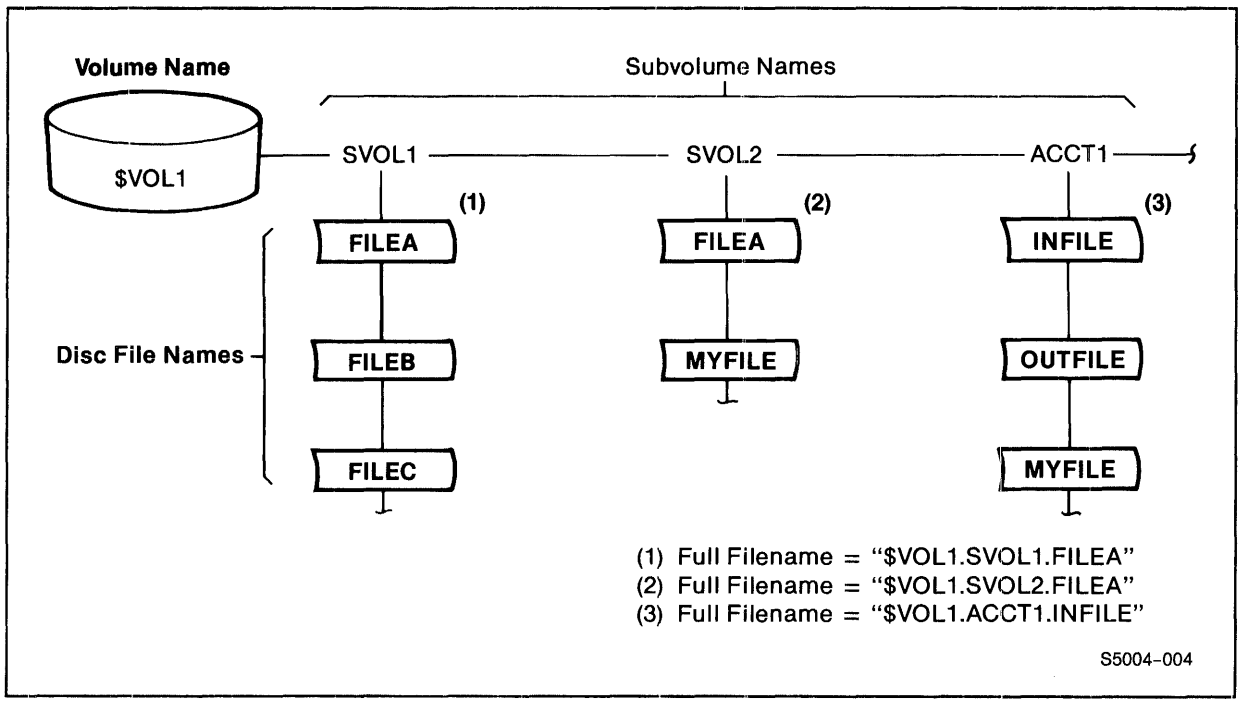


Figure 2-1. Disc File Organization

A disc file must be created before it can be accessed. A file is created by calling the file-system CREATE procedure or by using the CREATE command of the command interpreter. All file-system procedure call syntax is presented in the System Procedure Calls Reference Manual. The GUARDIAN command interpreter commands are presented in the GUARDIAN Operating System User's Guide.

A file can be designated as permanent or temporary. (A permanent file remains in the system when access is terminated; a temporary file is deleted.)

When you create a disc file, you must specify the maximum amount of physical disc space to be allocated for storing information. Physical space is allocated to files in the form of file extents that need not be contiguous. A file extent is a contiguous block of storage that can range in size from 2048 bytes to an entire volume. A file can have up to 16 extents (unless changed by SETMODE function 92--see the System Procedure Calls Reference Manual). The first extent is called the primary extent, and its size may be different from the other 15 secondary extents. File extents are allocated automatically by the file system as the need for space arises. Space not physically in use by one file can be 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 Incorporated.)

A disc drive having a removable pack 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 operator uses the RENAME command of the Peripheral Utility Program (PUP). Logical interlocks exist in the file system to ensure that an in-use volume cannot be demounted (this interlock can be overridden by the operator), 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.

Nondisc Devices

Nondisc 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 nondisc 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. See your System Management Manual for details.

What constitutes an input-output transfer with nondisc devices is dependent on the characteristics of the particular device. 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 nondisc devices are described in detail in sections 6 through 9.

Processes (Interprocess Communication)

A process (running program) can communicate with other processes through standard file-system data transfers. A communication can consist of a simple one-way transfer from the originating process to the destination process, or a two-way transfer where the originating process waits for a reply from the destination process. The information transferred because of interprocess communication appears identical to that of other files; there is no implicit or special data format.

A process is identified by its process ID as the destination of an interprocess communication. A process receives messages from other processes and from the operating system through a file identified by the name "$RECEIVE".

A process ID uniquely identifies a process. There are two mutually exclusive forms of the process ID: the timestamp form and the process-name form. Their formats are presented in Section 3. Process IDs are sometimes referred to as CRTPIDs.

A process identified by the timestamp form of the process ID is not known throughout the system. Rather, the process is known only by its creator and its immediate descendants (if any). Communication with processes identified by the timestamp form of process ID typically occurs only between processes having this creator-descendant relationship. This form of process ID is assigned to a process by the operating system when the new process is created. It consists of a timestamp of the time when the process was created, the number of the processor module where the process is executing, and a processor-local process number.

A two-way message in this environment is defined as the sending of a message from the originator (requester) to the server and the resultant reply by the server back to the originator, as shown in Figure 2-2.

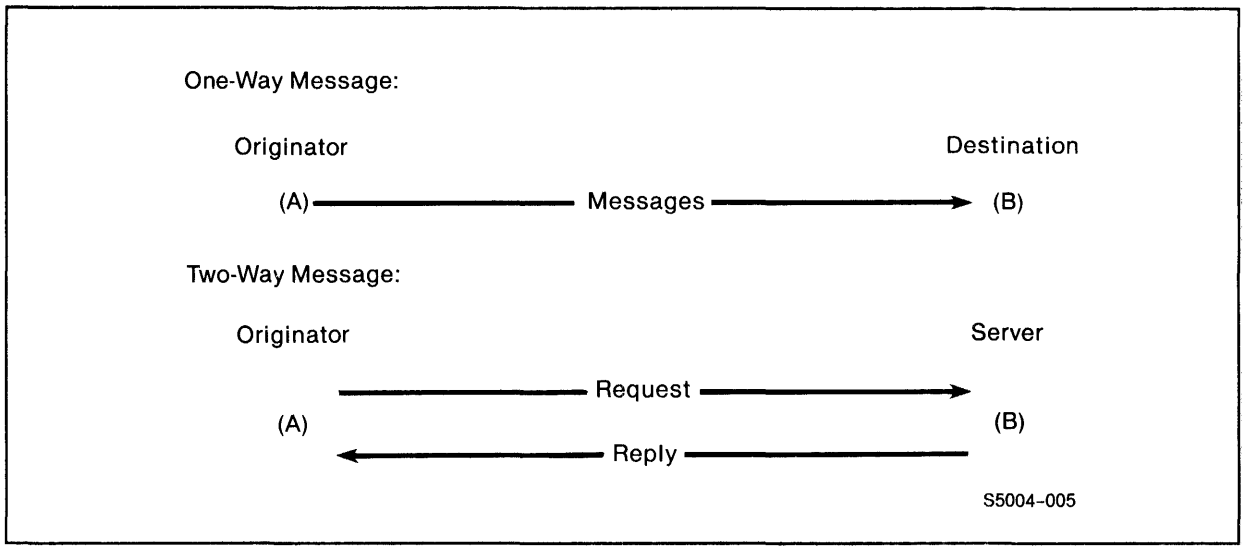


Figure 2-2. Communication With a Process by Process ID

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 (such as a line printer) are known throughout the system. If a process or 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").

As shown in Figure 2-3, there are certain fault-tolerant aspects involved when 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.

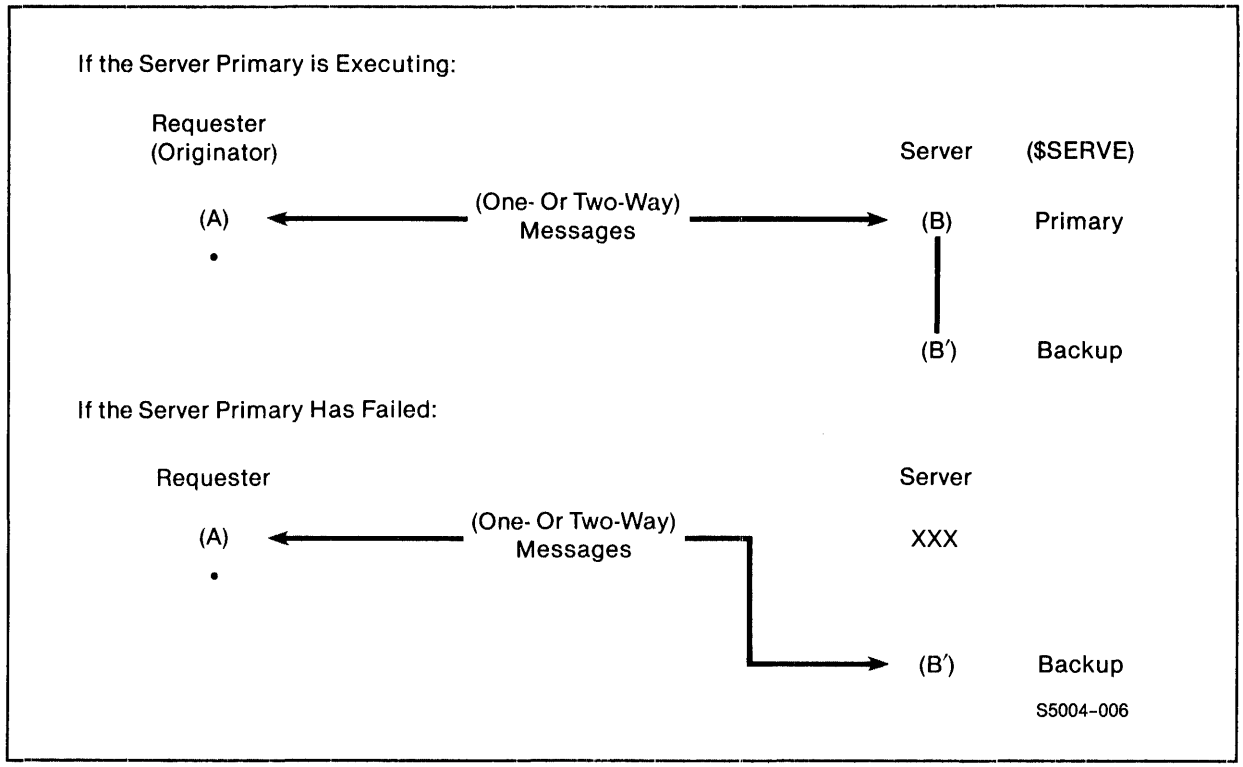


Figure 2-3. Communication With a Process Pair by Process Name

To receive and reply to communications from other processes and to receive messages from the operating system, a process references a file having the name "$RECEIVE" (there is only one $RECEIVE file per process). Communication with $RECEIVE is illustrated in Figure 2-4.

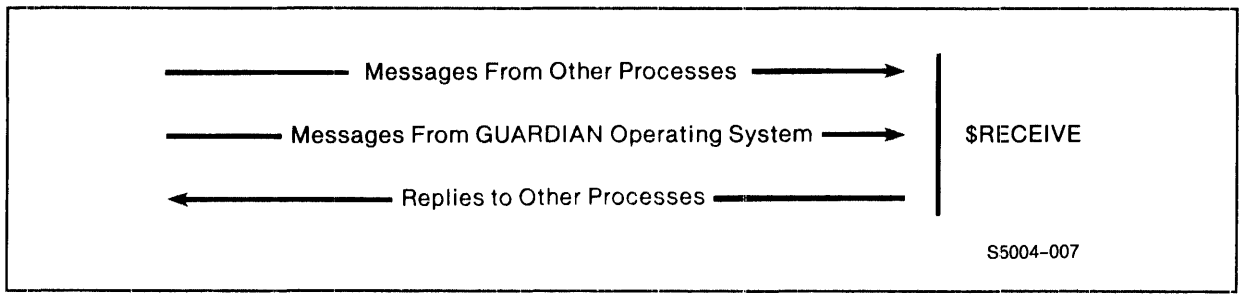


Figure 2-4. $RECEIVE File

Unlike disc files and nondisc devices, reading from the $RECEIVE file does not solicit input. Reading from $RECEIVE only checks or waits for an incoming 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 a reply is sent for a particular message, it is identified by passing the message's associated message tag back to the system. Communication between processes is described in greater detail in Section 4.

Communication between system processes and user processes occurs in a manner similar to that described above. System messages are simply interprocess messages sent from the operating system to the user process. These system messages should be read from $RECEIVE when a file-system error 6 (SYSTEM MESSAGE RECEIVED) is encountered.

Operator Console

A process may log messages on the operator console through the operator process referenced by the file name $0. The operator console is a write-only file. The current date and time and the ID of the process that logged the message are added as a prefix to console messages by the operating system. There is no special format imposed for logging messages on the operator console. Operations involving the operator console are described in Section 10.

FILE NAMES

File names are used to access devices, disc files, processes, and the operator console through the file-system OPEN procedure. A file name can be that of a disc device, nondisc device, or named process. File names are also used when creating new disc files, purging old disc files, and renaming disc files.

Disc File Names

Disc file names are stored internally in the form:

```
word: [0:3]              [4:7]               [8:11]
      $<volume-name>  <subvolume-name> <disc-filename>
```

Volume Name

Volume names identify disc packs (each pack in the system has a volume name). Volume names are assigned at system generation time and when new disc packs are introduced into the system. Their names are usually assigned by the system manager. A volume name must be preceded by a dollar sign ($). It consists of a maximum of seven alphanumeric characters; the first must be a letter. Alphanumeric means only A through Z and 0 through 9.

Subvolume Name

This name identifies a subvolume, which is a subset of the disc files on a volume (disc pack). Subvolume names are assigned programmatically when disc files are created. Anyone can create a new subvolume name or create new files within an existing subvolume. A subvolume name consists of a maximum of eight alphanumeric characters; the first character must be a letter.

File Name

Disc file names are assigned programmatically when disc files are created. A disc file name consists of a maximum of eight alphanumeric characters; the first character must be a letter.

Disc File Name Expansion (External File Name)

As an operating convenience, the GUARDIAN command interpreter accepts, where a file name is a parameter to a command, disc file names in partially-qualified form. As a minimum, a partial file name must consist of a <disc-file-name>. Partial file names are expanded to full file names according to the following rules:

1. If the <volume-name> is omitted from the external file name, the <default-volume-name> is used in its place.

2. If the <subvolume-name> is omitted from the external file name, the <default-subvolume-name> is used in its place.

A complete description of default file names appears in the GUARDIAN Operating System User's Guide.

Temporary File Name

This name identifies a temporary disc file. Temporary file names are assigned by the file-system CREATE procedure when temporary files are created. If you pass only a volume name followed by blanks to CREATE, it will create a temporary file name. A temporary file name consists of a number sign (#) followed by four numeric characters; for example:

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, such as "$STORE1_#0931 ", is returned from the call to 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 a letter. For example:

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

$0

$0 (dollar-zero) is a special file name used to write messages on the operator console; for example:

```
INT .FNAME[0:11] := ["$0", 11 * ["  "]];
```

The use of $0 is discussed more fully in Section 10, "Interfacing to the Operator Console".

$RECEIVE

$RECEIVE is a special file name used to receive and reply to messages from other processes; for example:

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

Internal File Names

There are two forms of file names--external and internal. The internal form is used within the system when passing file names between application processes and the operating system.

The internal form of a file name is shown in Figure 2-5.

```
<filename> ! 12 words, blank filled.

   to access permanent disc files, use

      <filename>[0:3]  =  $<volume-name>  <blank-fill>
      <filename>[4:7]  =  <subvolume-name>  <blank-fill>
      <filename>[8:11] =  <disc-filename>  <blank-fill>

   to access temporary disc files, use

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

   to access nondisc devices, use

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

   to communicate with other processes, use

      <filename>[0:11] =  $RECEIVE <blank-fill>

         to perform READ, READUPDATE, and REPLY operations,

      <filename>[0:11] =  <process-id>  <blank-fill>

         to perform WRITE and WRITEREAD operations

   to write on the operator console, use

      <filename>[0:11] =  $0 <blank-fill>
```

Figure 2-5. Internal Form of a File Name

External File Names

File names must be alphanumeric. This means they are made up of only the letters A through Z and the digits 0 through 9. Lowercase letters used to enter file names are converted to uppercase. The first character of a file name must be a letter.

File names can be preceded by a special character, such as a dollar sign ($) or a back slash (\).

File names are limited in length depending upon their use.

The file name of a nondisc device is represented to the command interpreter in the same form as the file system's internal representation; that is, as the device name or logical device number preceded by a dollar sign ($). These are presented in this manual in the form:

```
$<device-name>

      or

$<ldev-number>
```

The external form of the file name is used when entering file names into the system from the outside world (for example, by a user to specify a file name to the command interpreter).

The external form of a file name is shown in Figure 2-6.

The various forms of external file names are discussed in the GUARDIAN Operating System User's Guide.

Like the internal representation of a disc file name, the form accepted by the command interpreter for a disc file consists of three parts: a volume name, a subvolume name, and a disc file name. However, unlike the fixed-field representation of the internal form (where each part of a file name must begin in a specific position), disc file names are represented to the command interpreter (as well as to all other Tandem software programs) with the three parts separated by periods and concatenated into a contiguous string:

```
$<volume-name>.<subvolume-name>.<disc-filename>
```

The following example illustrates a disc file name:

```
$STORE1.ACCTRCV.SORTFILE
```

```
<external-filename>     ! up to 26 bytes.

to access permanent disc files, use

   [$<volume-name>.] [<subvolume-name>.]
     <disc-filename>  <delim>

to access nondisc devices, use

   $<device-name> <delim>     or

   $<ldev-number>  <delim>

to specify a process, use

   $<process-name>  <delim>

 <delim> is a delimiter; it can be any character that is not
 valid as part of an external file name, such as a blank.
```

Figure 2-6. External Form of a File Name

The conversion of file names from external form to internal form is performed automatically by the command interpreter for the IN and OUT file parameters of the RUN command (refer to the GUARDIAN Operating System User's Guide.

For general conversion of file names from the external to the internal form, the FNAMEEXPAND procedure is provided. Conversion of file names from the internal to the external form is done using the FNAMECOLLAPSE procedure. These procedures are described in the System Procedure Calls Reference Manual.

Correspondence of External to Internal File Names

The correspondence of the external form of file names to the internal form is shown in Figure 2-7.

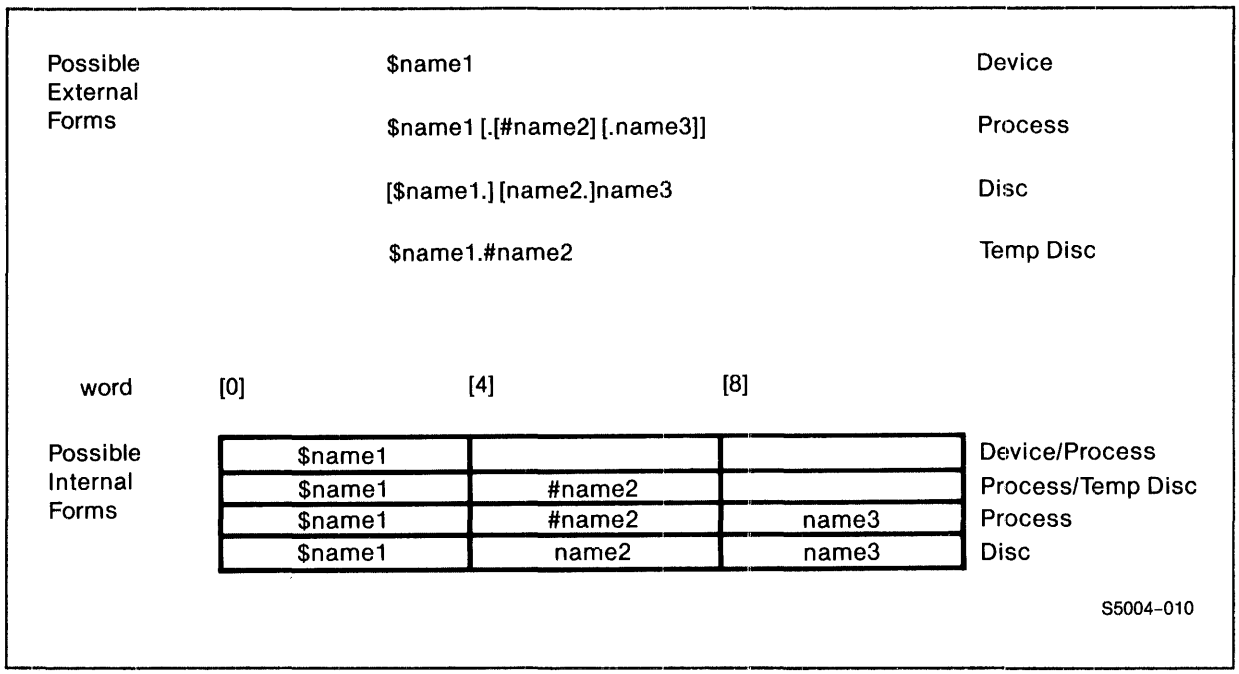


Figure 2-7. Correspondence of External to Internal File Names

When the external form of a file name is entered as a parameter to a command (for example, the IN parameter of the command interpreter RUN command), it is converted to the internal form as shown in Figure 2-7. For example, in the case of the IN parameter <filename>, the <filename> is converted from the external form to the internal form (and expanded if necessary) and sent to the application process in an interprocess startup message.

NETWORK FILE NAMES

File names can optionally include a system number that identifies a file as belonging to a particular system on a network. (A fuller description of file names appears in the GUARDIAN Operating System User's Guide. Information regarding networks of Tandem systems is presented in the EXPAND Reference Manual.)

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 is the network form.

Internal Network File Names

The internal form of a network file name is shown in Figure 2-8.

```
<network-filename>  ! 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

   <system-number>

      is an integer between 0 and 254 that designates a
      particular system.  The assignment of system numbers
      is made during SYSGEN.

   <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, described earlier in this section.
```

Figure 2-8. Internal Form of a Network File Name

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 mentioned in Figure 2-8.

External Network File Names

For the purpose of providing access to files on remote systems in a network, any file name can be qualified by a system name. (System names, and networks in general, are discussed in the GUARDIAN Operating System User's Guide. Additional information on network file names appears in the EXPAND Reference Manual).

A system name consists of a backslash (\) followed by up to seven alphanumeric characters, the first of which must be alphabetic. Any file name can be preceded by a system name.

The external form of a network file name is shown in Figure 2-9.

```
\<system-name>.<external-filename>


   <external-filename>

      is the external form of any legal file name.  The
      length of a device or process name used with a system
      name contains one character less than usual:  device
      names have at most six letters and/or digits, and
      process names have at most four letters and/or digits.
      For example:

   \NEWYORK.$SYSTEM.SYSTEM.MYFILE  ! fully qualified
                                   ! disc file name

   \REMOTE.$WXYZ  ! process

   \DETROIT.$SYSTEM.#1234  ! temporary disc file name
```

Figure 2-9. External Form of a Network File Name

Default System

Each command interpreter running on a system in a network has associated with it a default system that is used in file name expansion. When you log on, your default system is always the system on which your command interpreter is running. The default system can be changed with the SYSTEM command. See the GUARDIAN Operating System User's Guide for details.

Expansion of Network File Names

File names presented in external form to a command interpreter (or any other Tandem subsystem, such as EDIT or FUP) running on a system in a network are expanded using the default volume, subvolume, and system names. For example, assume that your current defaults are as follows:

```
default volume:      $MYVOL
default subvolume:   MYSUBVOL
default system:      \CALIF
```

If you enter this file name,	it is expanded to this form.
MYFILE	\CALIF.$MYVOL.MYSUBVOL.MYFILE
\NEWYORK.MYFILE	\NEWYORK.$MYVOL.MYSUBVOL.MYFILE
$PROC	\CALIF.$PROC

Correspondence of Internal to External Network File Names

When transforming an external file name to an internal one, the system replaces the system name with the corresponding system number. External network file names supplied as IN or OUT files in a RUN command are converted to internal form by the command interpreter before being passed to the new process. Thus an application process that reads its startup message and opens its IN file need not do anything different when remote files are involved.

NOTE

When used across a network, the length of a device or process name used with a system name must be one character shorter than usual to allow for the embedded backslash prefix (\): device names allow at most six alphanumeric characters, and process names allow at most four alphanumeric characters.

LOGICAL DEVICE NUMBERS

Logical device numbers identify entries in an internal operating system table which, in turn, identify particular input-output devices in the system. Logical device numbers are assigned to physical I/O devices at system generation. A logical device number must be preceded by a dollar sign ($). It must consist of a maximum of four numeric characters; the maximum logical device number is 4095.

PROCESS IDs AND PROCESS NAMES

A process is uniquely identified by its process ID. There are three equivalent forms of the process ID: the timestamp form, the process-name form, and the network form. Process names are known throughout the system by means of the destination control table (DCT).

Timestamp Form of Process ID

For the timestamp form, the GUARDIAN operating system assigns the process ID when the process is created. The form of this type of process ID is:

```
<process-id> [0].<0:1>  = 2
<process-id> [0].<2:7>  = unused
<process-id> [0].<8:15> = <system number> is 0 through 254
<process-id> [1:2]      = low-order 32 bits of creation
                          timestamp
<process-id> [3].<0:3>  = unused
<process-id> [3].<4:7>  = <cpu> where process is executing
<process-id> [3].<8:15> = <pin> assigned by operating system
                          to identify the process in the CPU
```

Process-Name Form of Process ID

For this form, the process ID contains an application-defined <process-name>. The process name is specified before process creation time, then entered into the DCT at process creation time.

The general form of this type of process ID is:

```
<process-id>[0:2] = $<process-name>
<process-id>[3]   = <(two blanks)> or <cpu,pin>
```

<process-name> must be preceded by a dollar sign ($) and consist of a maximum of five alphanumeric characters; the first character must be a letter. The <cpu,pin> may be included but is ignored. However, if it is included, it must be valid.

If a process name represents a process pair and the process accessing the pair is not a member of the pair, then the process name references the pair as a single entity. Communication occurs with the primary process of the pair while it is operable. If it becomes inoperable, communication is redirected to the backup process (in a manner invisible to processes outside of the pair).

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.

Network Form of Process ID

The network form of the process ID is:

```
<process-id>[0].<0:7>     = "\" (ASCII backslash)
<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 the process name in words 1 and 2 can contain at most four alphanumeric characters (the first one must be a letter, as usual) and does not include the initial dollar sign ($).

The application program rarely, if ever, concerns itself with octal system numbers in network file names. Usually, the application passes the external form of the file name (which contains a system name, rather than a number) to the procedure FNAMEEXPAND, which converts the system name into the corresponding number.

The external form of network file names is described later in this section. Information and examples regarding the use of network file names in operating and programming Tandem systems can be found in the EXPAND Reference Manual. Conversion between internal and external forms of network file names is accomplished by the procedures FNAMEEXPAND and FNAMECOLLAPSE.

The following process control procedures relate to process IDs:

- 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.

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 internal process file names of the form shown in Figure 2-10.

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

   #<1st-qualif-name>

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

   <2nd-qualif-name>

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

Figure 2-10. Internal Process File Names Form

Only the process name has meaning to the file system (it indicates the particular process or process pair being opened). The qualifier names have no particular meaning to the file system. They are, however, checked for proper format. Instead, their meaning must be interpreted by the process being opened, which receives the qualifier names as part of the OPEN system message.

HOW TO ACCESS FILES

Communication between an application process and a file is established through the file-system OPEN procedure. An array in the application process data area, containing the internal-form 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 accessing it through subsequent system procedure calls.

For example, to establish communication (open a file) with a terminal referenced by the device name "$TERM1", you can use the following 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);
    .
```

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

To write (output) to a file, the file number returned from OPEN is passed as a parameter to the WRITE procedure:

```
    .
    .
loop:
    .
CALL WRITE(FILENUM,BUFFER,72);
    .
```

This WRITE causes 72 bytes of the array BUFFER to be printed on the terminal.

To read (input) from a file, the file number returned from OPEN is passed as a parameter to the file-system READ procedure:

```
    .
    .
CALL READ(FILENUM,BUFFER,72,NUMXFERRED);
    .
```

This READ permits up to 72 bytes to be input from the terminal into the array BUFFER. A count of the number actually input is returned in NUMXFERRED.

```
   .
   GOTO loop;
```

The communication link with a file is terminated through use of the file-system CLOSE procedure:

```
   .
   CALL CLOSE(FILENUM);
      .
```

The file representing the terminal is closed.

Disc Files

Disc files must be created before access is possible. Creation is accomplished by calling the file-system CREATE procedure:

```
   INT .DISC^FNAME[0:11] := "$VOL1   MYFILES FILEA   ";

   .
   CALL CREATE(DISC^FNAME);
      .
```

This creates a disc file with the subvolume name "MYFILES" and the disc file name "FILEA" on the disc volume identified as "$VOL1". A primary and secondary extent size of 2048 bytes and a file code of "0" is implied.

```
   CALL OPEN (DISC^FNAME,FILENUM);
      .
```

This opens the disc file referenced by the file name 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);
    .
```

This 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);
```

This 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);
    .
```

This positions the file pointers to point at relative byte 4096.

```
    .
CALL READ(FILENUM,BUFFER,512,NUMXFERRED);
    .
```

This transfers 512 bytes from the disc file starting at relative byte 4096 into BUFFER. The next-record pointer is incremented by 512 so that further sequential access is automatic. The current-record pointer now points at relative byte 4096.

Data can be written at the position indicated by the current-record pointer through use of the WRITEUPDATE procedure. Using the position of the preceding example, the call

```
CALL WRITEUPDATE (FILENUM, BUFFER, 512);
```

This writes 512 bytes of the array BUFFER starting at relative byte 4096.

Terminals

Operations with terminals tend to be of the form "write, then read". The WRITEREAD procedure combines these two operations. A special hardware feature incorporated in the asynchronous multiplexer controller ensures that the computer system is immediately ready for input following the write of the prompt.

For example, assume that FILENUM contains a file number representing a terminal:

```
    .
BUFFER ':=' "PLEASE INPUT ACCOUNT NUMBER";
CALL WRITEREAD(FILENUM, BUFFER,27,72,NUMXFERRED);
    .
```

This writes 27 bytes of the array BUFFER as a prompt, then prepares for reading up to 72 bytes from the terminal back into BUFFER. A count of the number of bytes input is returned in NUMXFERRED.

Processes

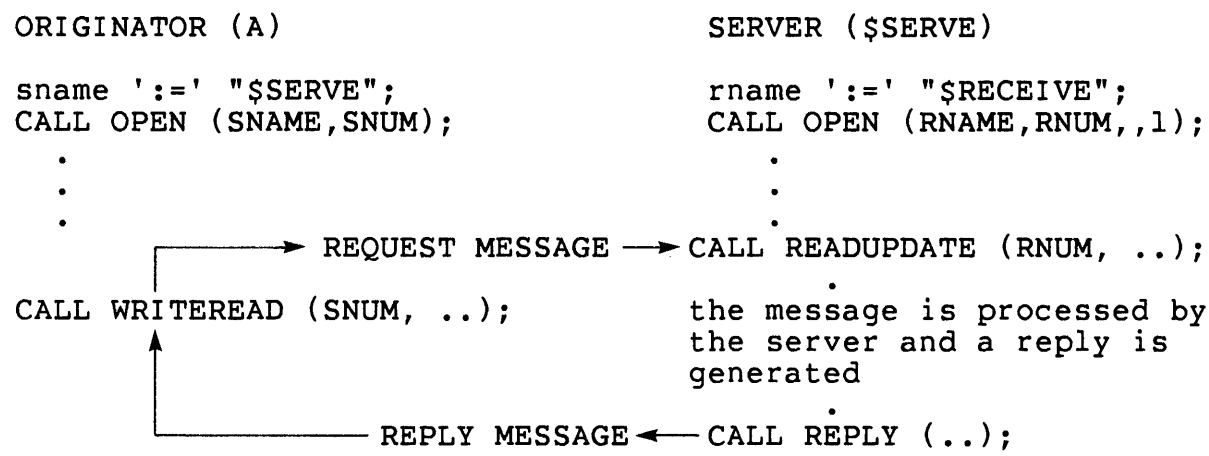
A process can write messages to another process by opening a file that references the process by its process ID (the process ID can be of the process-name form or the timestamp form). A process reads messages from other processes by opening a file designated $RECEIVE. (System messages from the operating system can be also be read in this manner if the proper OPEN flags parameter bit is set.)

A one-way message can be sent to a process identified by the timestamp form of process ID and received by that process as follows:

```
ORIGINATOR (A)                          DESTINATION (B)

                                        rname ':=' "$RECEIVE";
CALL OPEN (PIDB,BFNUM);                 CALL OPEN (RNAME,RFNUM);

    "PIDB" contains B's process ID
  .                                         .
  .                                         .
CALL WRITE(BFNUM,..); ---->MESSAGE ---->CALL READ(RFNUM,...);
```

A sends a message to B by way of B's process ID. B reads the message by use of its $RECEIVE file.

A two-way message can 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 by a call to READUPDATE. When the reply is ready, it is sent back to A by a call to REPLY. When A receives the reply, WRITEREAD completes, and A resumes processing.

COORDINATING MULTIPLE FILE ACCESSORS

A file can be accessed by several different processes at the same time. To coordinate such 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 to be performed by an accessor. The access mode is specified as either read-write (default access mode), read-only, or write-only.

The exclusion mode specifies how much access other processes will be allowed. It can provide shared, exclusive, or protected access.

- Shared access, the default exclusion mode, indicates that the opening process can allow simultaneous read and/or write access by other processes to the file.

- Exclusive access indicates that the opening process cannot allow any simultaneous access of any kind to the file. Therefore, any further attempts to open the file, while the file is open, are rejected. Likewise, if a file is already open, any attempt to open the file with exclusive access is rejected.

- Protected access indicates that the opening process can allow simultaneous read access to the file but cannot allow simultaneous write access to the file. Therefore, while the file is opened with protected access, any further attempts to open the file with read-write or write-only access mode are rejected. Likewise, if the file is already open with read-write or write-only access mode, any attempt to open it with protected access is rejected. However, simultaneous accessors can open a file with read-only access mode.

An additional method of access coordination is provided for disc files through the LOCKFILE and UNLOCKFILE procedures. Multiple processes accessing the same disc file call LOCKFILE before performing a critical sequence of operations to that file. If the file is not currently locked, it becomes locked, and the process continues executing. This prevents other accesses to the file until it is unlocked through a call to UNLOCKFILE. If the file is locked, a caller of LOCKFILE is suspended until the file is unlocked. If a process attempts to write to a locked file, the access is rejected with a "file is locked" error indication; if a process attempts to read from a locked file, it is suspended until the file is unlocked.

An alternate mode for file locking is provided. Instead of suspending the caller to LOCKFILE if the requested file is locked, the lock request is rejected, and the call to LOCKFILE completes immediately with a "file is locked" error indication. Moreover, if a process attempts to read from a locked file, the read is immediately rejected. The alternate locking mode is specified by a call to the SETMODE procedure.

File locking is described in the ENSCRIBE Programming Manual.

WAIT I/O AND NOWAIT I/O

The file system can allow an application process to execute concurrently with its file operations when you specify "nowait" I/O.

The default is "wait" I/O; when designated file operations are performed (using system procedure calls), the application process is suspended, waiting for the operation to complete.

Nowait 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 system procedure call.

Each time a file is opened, the opener specifies whether wait or nowait I/O is to be in effect when designated file operations are performed. If nowait I/O is specified, then the maximum number of concurrent operations to be permitted must also be specified when the file is opened. Disc files are limited to one concurrent operation (one outstanding nowait call) per file-opening.

For example, to open a file so that one concurrent file operation is permitted (a "nowait" file), this call could be included in an application program (assume that file identifies a valid file):

```
CALL OPEN (FILE, FILE^NUMBER^1, 1);
```

The third parameter, "1", specifies that one concurrent operation is permitted. (This parameter is also used for other purposes; see the description of the OPEN procedure in the System Procedure Calls Reference Manual.)

Any input-output operation involves initiation and completion. With wait files, initiation and completion are both performed in the same system procedure call. For example, on a wait file, the call

```
CALL READ (FILE^NUMBER^0, BUFFER, ..);
```

initiates the I/O operation, then the application process is suspended, waiting for its completion.

With nowait files, the initiation is performed in one call:

```
CALL READ (FILE^NUMBER^1, BUFFER, ..);
```

After this call initiates the I/O operation, process execution continues concurrently with the I/O transfer. Later, the operation is completed by another call:

```
CALL AWAITIO (FILE^NUMBER^1, ..);
```

If the I/O operation is not complete when AWAITIO is called, the process is suspended until completion occurs or an application-defined timeout expires.

Multiple operations (with multiple files) can be in progress simultaneously. Concurrent operations associated with separate file-openings are completed as they finish. Concurrent operations associated with a particular file-opening also are completed as they finish, unless SETMODE 30 is used (refer to the AWAITIO procedure considerations in the System Procedure Calls Reference Manual.

When a record is inserted into a file having an alternate key, the whole operation is wait I/O, even if you specify nowait I/O. This is because (1) only the last I/O can be nowait, (2) the primary-file insertion is always done first, and (3) alternate-key-file insertions are always wait I/O. For deletions and updates, on the other hand, the alternate-key file is always modified first, so the primary-file modification can be nowait I/O.

The difference between wait and nowait I/O is illustrated in Figure 2-11. The action of nowait I/O during multiple concurrent operations is shown in Figure 2-12.

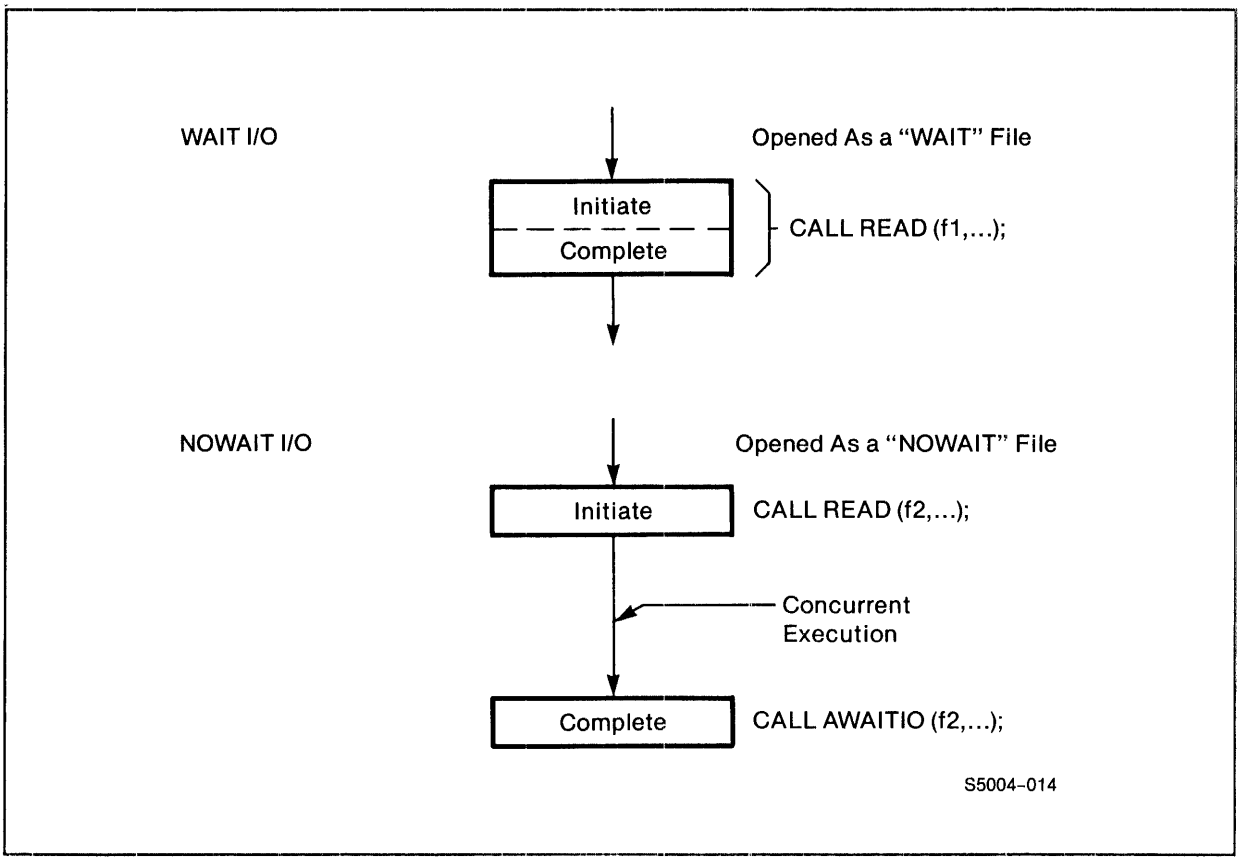


Figure 2-11. Wait I/O Compared With Nowait I/O Operation

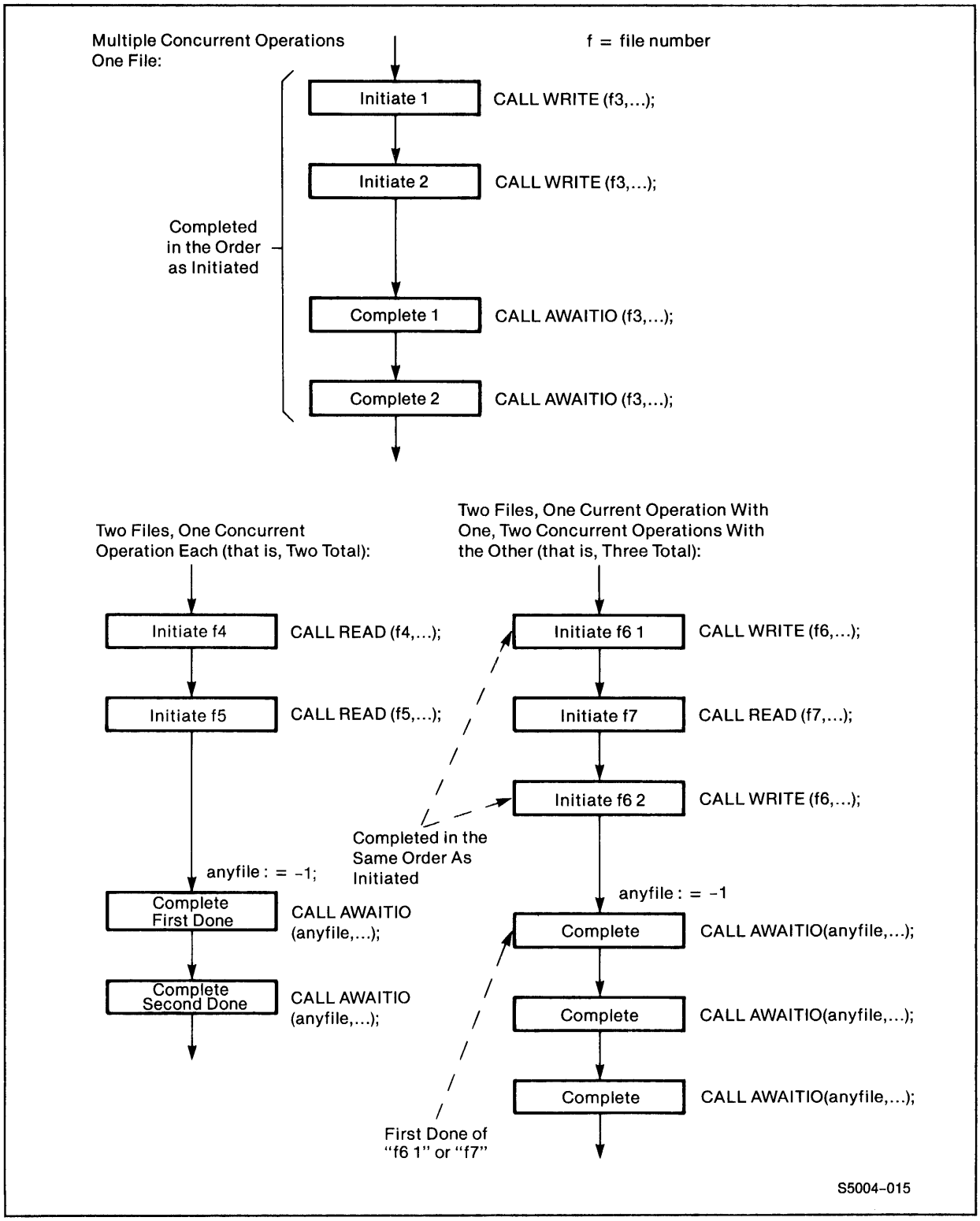


Figure 2-12. Nowait I/O (Multiple Concurrent Operations)

HOW THE FILE SYSTEM WORKS

This description introduces the internal operation of the file system. In particular, every programmer should have a thorough understanding of the action that the file system takes when a communication-path failure occurs and the corresponding action that the application program must take to recover. This discussion includes:

- Hardware I/O structure
- Software file system
- Opening files
- File transfer
- Closing files
- Automatic path error recovery for disc files
- Mirrored volumes

Hardware I/O Structure

The hardware structure of Tandem systems is designed so that two physically independent communication paths exist between any application process and any I/O device.

The hardware communication path associated with an I/O operation includes:

- The interprocessor buses
- The processor module controlling the device
- The I/O channel to which the device is connected
- The I/O controller

Interprocessor buses carry data and control information between processor modules. The interprocessor bus is not part of the communication path if the processor module controlling the device is the same one where the application process requesting an I/O operation is running.

The processor module controlling a device executes I/O instructions to command the device to perform designated I/O functions, contains the main memory where the I/O transfer takes place, and receives the completion status from the hardware controller.

The I/O channel carries the control and data signals between a processor module and I/O controllers. (As many as 32 controllers can be connected to a single channel.)

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.

The existence of two physically independent communication paths is achieved 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. Thus, if one channel fails, control of the I/O controller is assumed by the I/O channel connected to the other processor module.

The hardware I/O structure is depicted in Figure 2-13. The System Description Manual for your system contains more detailed information.

Software (File System)

If at any time during a file operation any part of a communication path fails, the file operation can still be completed successfully.

The file system is an integral part of the GUARDIAN operating system. A copy of the operating system resides in each processor module in the system. Each copy contains only what is necessary to control the input-output devices connected to its particular processor module.

System control of I/O devices is accomplished by means of system I/O processes. The action of an I/O process is to accept a request from the file system (the request initially comes from an application process), perform the requested action (read or write), return the completion status of the operation (and also data if a read operation) to the file system, then wait for another request.

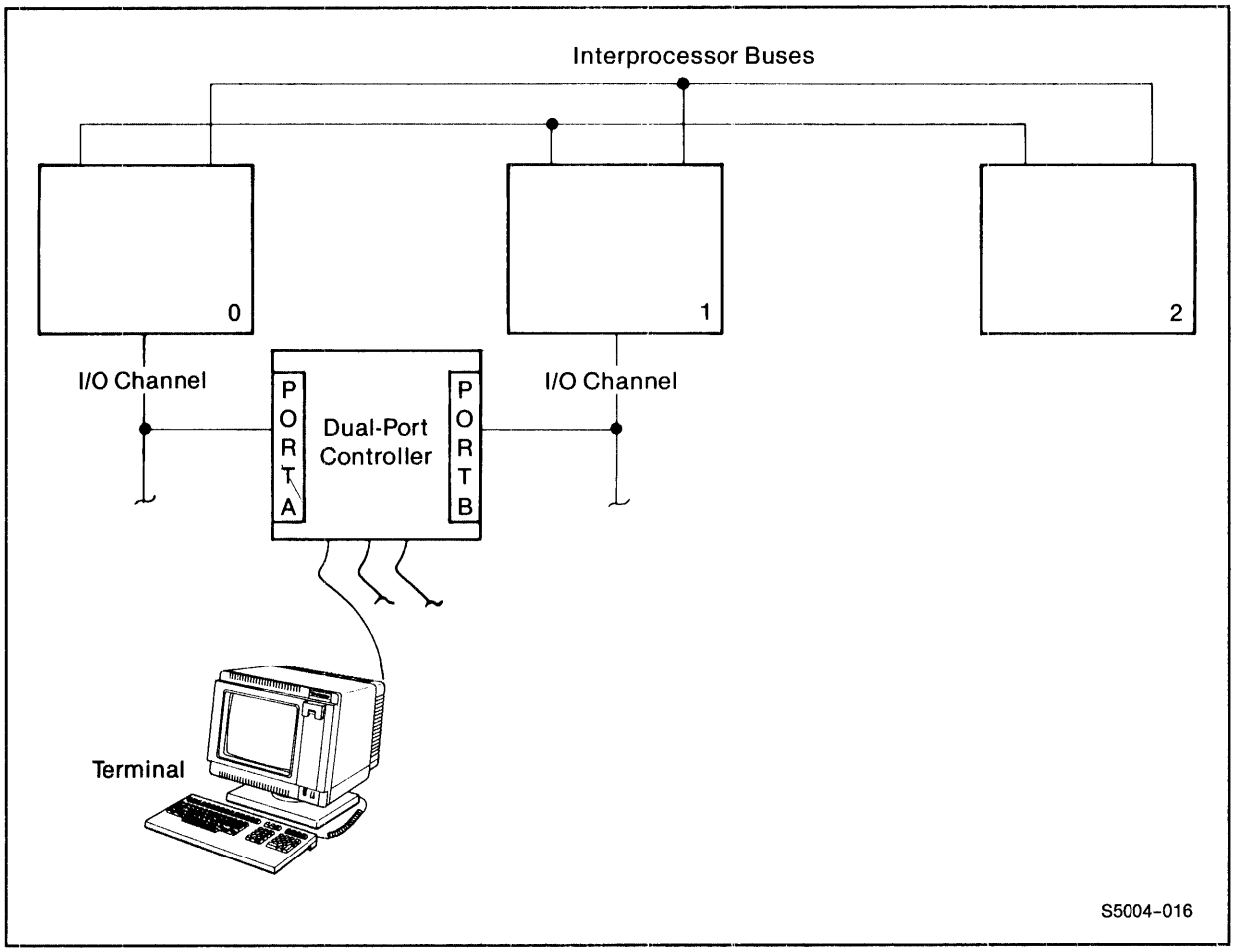


Figure 2-13. Hardware I/O Structure

There are two system I/O processes for each device (or set of devices in the case of terminals or data communication lines), one in each of the two processors that are physically connected to a given device. One process is designated the primary I/O process; the other is designated the backup I/O process. (This primary or backup designation is made during system generation.) Either I/O process is capable of controlling the device. However, they do not control the device simultaneously. Instead, the primary I/O process controls the device exclusively and, at the same time, keeps the backup I/O process informed (with checkpoint messages) of the activity on the device.

The communication path (processor module, I/O channel, and controller port) through a primary I/O process to the device that it controls is called the primary path; the path through a backup I/O process is called the alternate path. If the file

system (or the operating system on behalf of the file system) detects a failure in the primary path, it shuts down the primary path and automatically reroutes subsequent communication to the device across the alternate path. The backup I/O process takes control of the device and, in fact, becomes the primary I/O process for the device.

In the case of disc files, error recovery following a failure of the primary path is automatic, and this type of failure is completely invisible to the application program (see "Automatic Communication-Path Error Recovery for Disc Files" later in this section). In the case of nondisc devices, a path error indication is returned to any application processes that were performing I/O with the device when the failure occurred. The application program then has the responsibility of retrying the file operation following this type of failure (see "Error Recovery" in this section).

Figure 2-14 depicts the primary and alternate communication paths to a device. While the primary path is operable, all I/O transfers occur along 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.

When the original primary path is restored to system operation, it becomes the current backup path. The original primary path is restored to primary operation only when: the system is cold loaded, a failure occurs in the current primary path, a PUP PRIMARY command is executed to switch control of the device, or RETURN TO CONFIGURED PRIMARY is configured for the device and the original primary processor module is reloaded. (See your System Management Manual for an explanation of "cold load" and "reload".)

Executing System Procedures

System procedures reside in operating system code but execute in the application process's environment. When a file-system procedure (or any 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 2-15. The maximum amount of local storage required by a call to a system procedure is approximately 400 words.

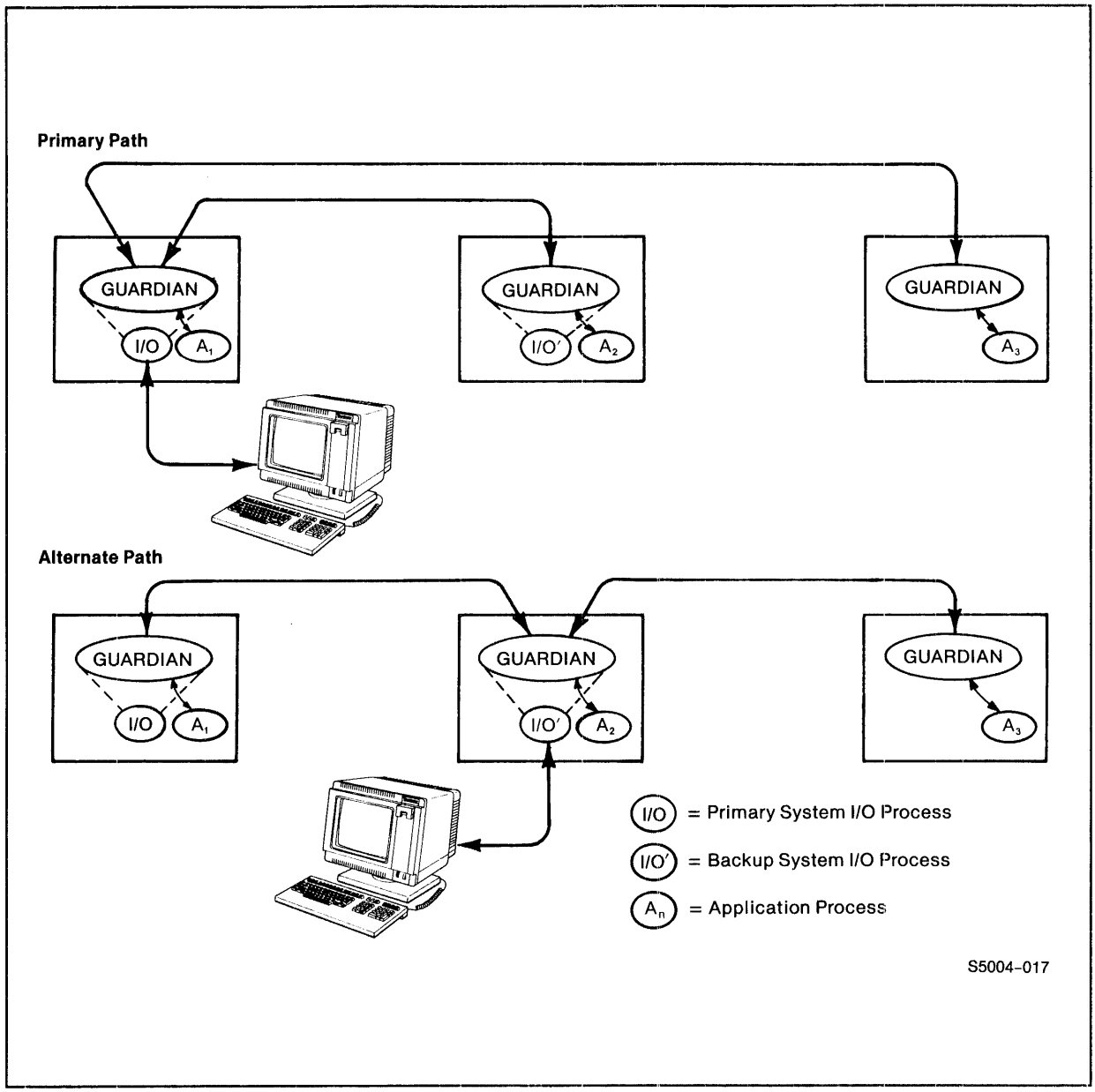


Figure 2-14. Primary and Alternate Communication Paths

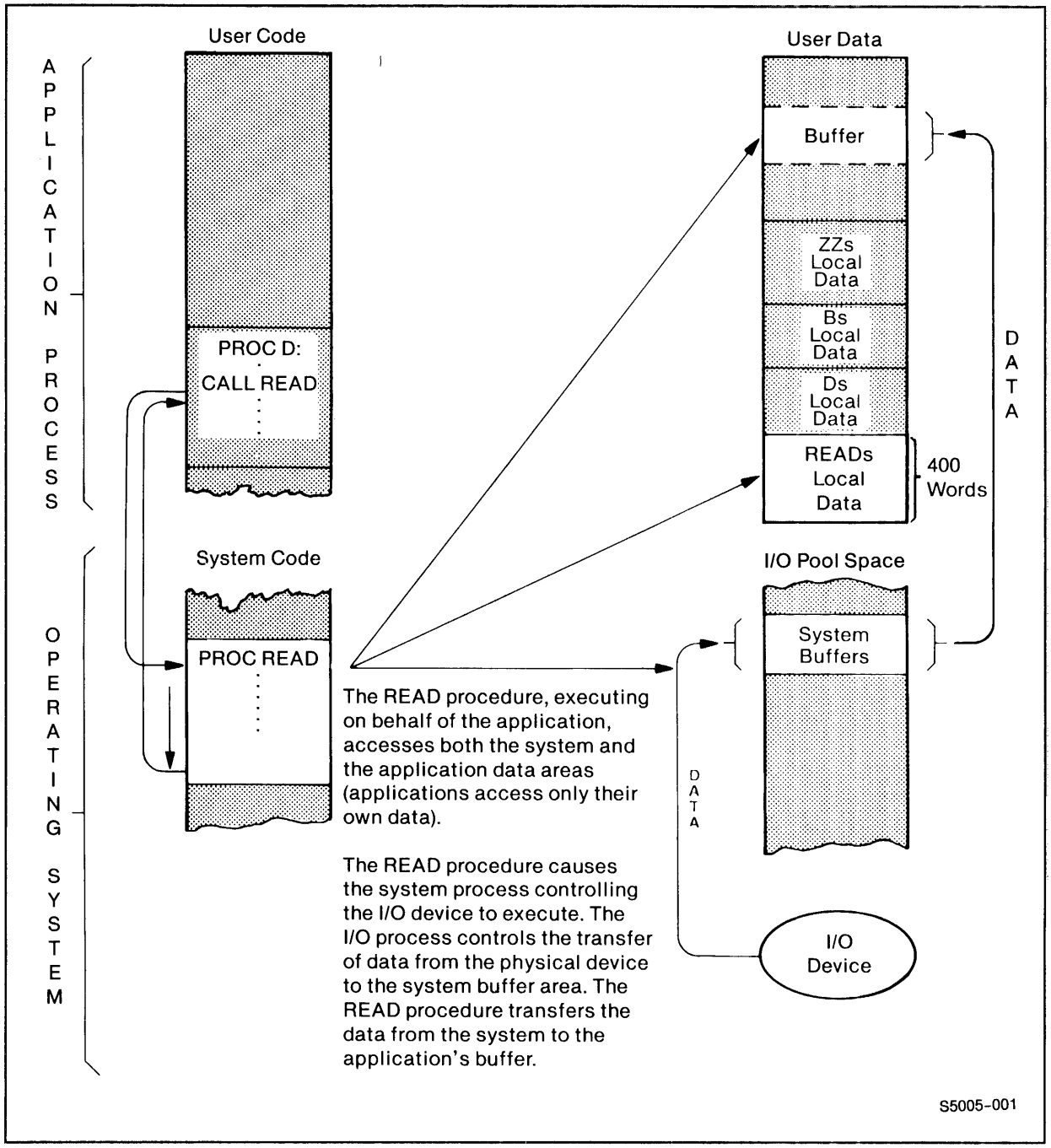


Figure 2-15. System Procedure Execution

Opening Files

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. This 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 or volume, and the process ID of the backup system I/O process that controls the device or volume.

In Figure 2-16, the destination control table is searched for an entry corresponding to the volume name "$VOLUME". The entry is associated with logical device 4, and a path is established to the primary I/O process controlling the device.

Next, in the case of disc files, the I/O process searches a directory on the disc volume for the subvolume name and the disc file name that was supplied in the OPEN procedure. The entry associated with a subvolume and disc file name is a file label that contains information describing the state of the file, including the location of allocated extents, end-of-file location, file type, and so on.

In Figure 2-16, the file label is searched for a subvolume designated "MYFILES" and a disc file named "FILEA".

Once the file is located, whether it is a disc file, nondisc device, process, or the operator console, an access control block (ACB) is created for that file in the memory of the processor where the caller to OPEN is running. The ACB is used by other file system functions when referencing the file. It contains information such as the logical device number of the device where the file resides and, for disc files, information local to the particular open of the file, such as the current-record pointer and the next-record pointer.

If the open is to a disc file and the file is not currently open, one file control block (FCB) is created in the memory of each of the two processor modules that contain the system I/O processes that control the volume containing the file. The FCB contains information that is global to all accessors of the file. This includes a copy of the information from the file label, such as allocated extents and end-of-file location, along with dynamic control information such as which process has the file locked and which processes, if any, are waiting to lock the file.

There is a single FCB for each open disc file in the system (in each of the two processor modules controlling the associated device), whereas an ACB is created every time a file is opened. Thus each opening of a given file provides a logically separate

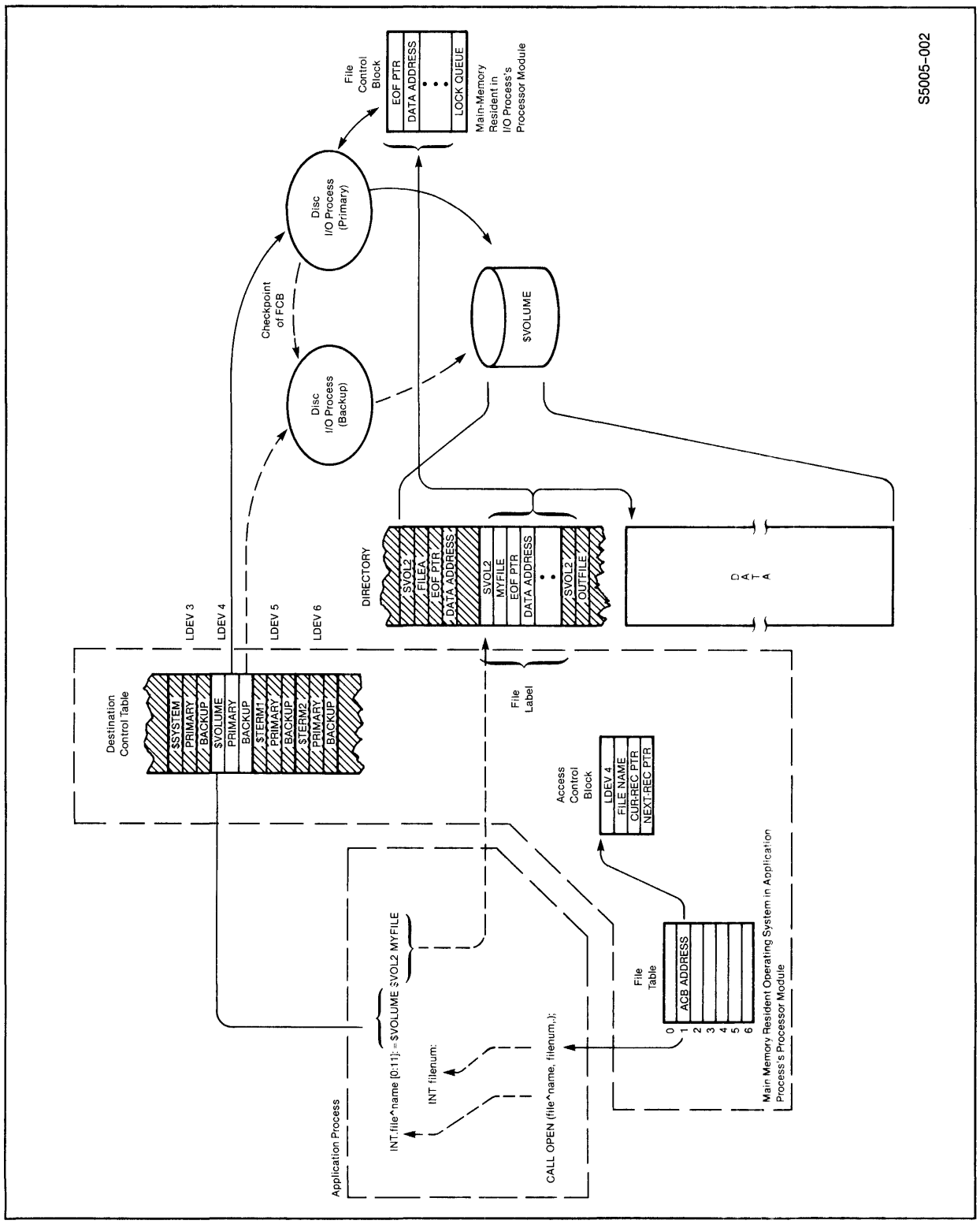


Figure 2-16. Opening a File

access to that file (separate current-record and next-record pointers), yet the end-of-file location (maintained in the FCB) has the same setting for all accessors of the file.

In Figure 2-16, the ACB indicates that logical device 4 is associated with the disc file indicated by "$VOL1".

When OPEN completes, it returns a file number to the application process. The file number is an index into a table that contains an address pointer to the associated ACB.

File Transfers

As previously mentioned, the file number returned from OPEN is used by system procedures to access an open file. The file number can be thought of as a pointer to an access control block (ACB). When performing an I/O operation, the file number is used to locate an ACB, which in turn provides a logical device number that is then used as an index into the DCT. The corresponding entry in the destination control table provides the process ID associated with the primary path to the physical I/O device.

In Figure 2-17, the ACB indicates that the device is logical device 4.

For disc files, the information in the ACB (such as the current-record and next-record pointers) and the information in the FCB (such as the end-of-file pointer and addresses of allocated extents) is updated with the execution of each I/O operation.

As file accesses requiring changes to the FCB are made, the system process currently responsible for controlling the disc ensures that the copy of the FCB in the other processor that can access the disc is updated. Thus, if the primary processor fails, the backup has all the information necessary for a smooth transition (invisible to the user). In addition, when a new extent is allocated or the file is renamed, the file label on the disc is updated to reflect this change. This ensures that no disc space is lost, even in the event of a total system failure. However, when the end-of-file (EOF) pointer is changed or the file is written into, which requires updating the last modification timestamp, only the main-memory copies are updated. (Updating the file label each time the file is written into would be an unnecessary amount of additional overhead because the current EOF and last modification timestamp would be lost only if a total system failure occurs. The user who is concerned about the EOF being updated on disc can force this to happen with the CONTROL request to set the EOF, or by using the FUP command to set REFRESH on for the file.)

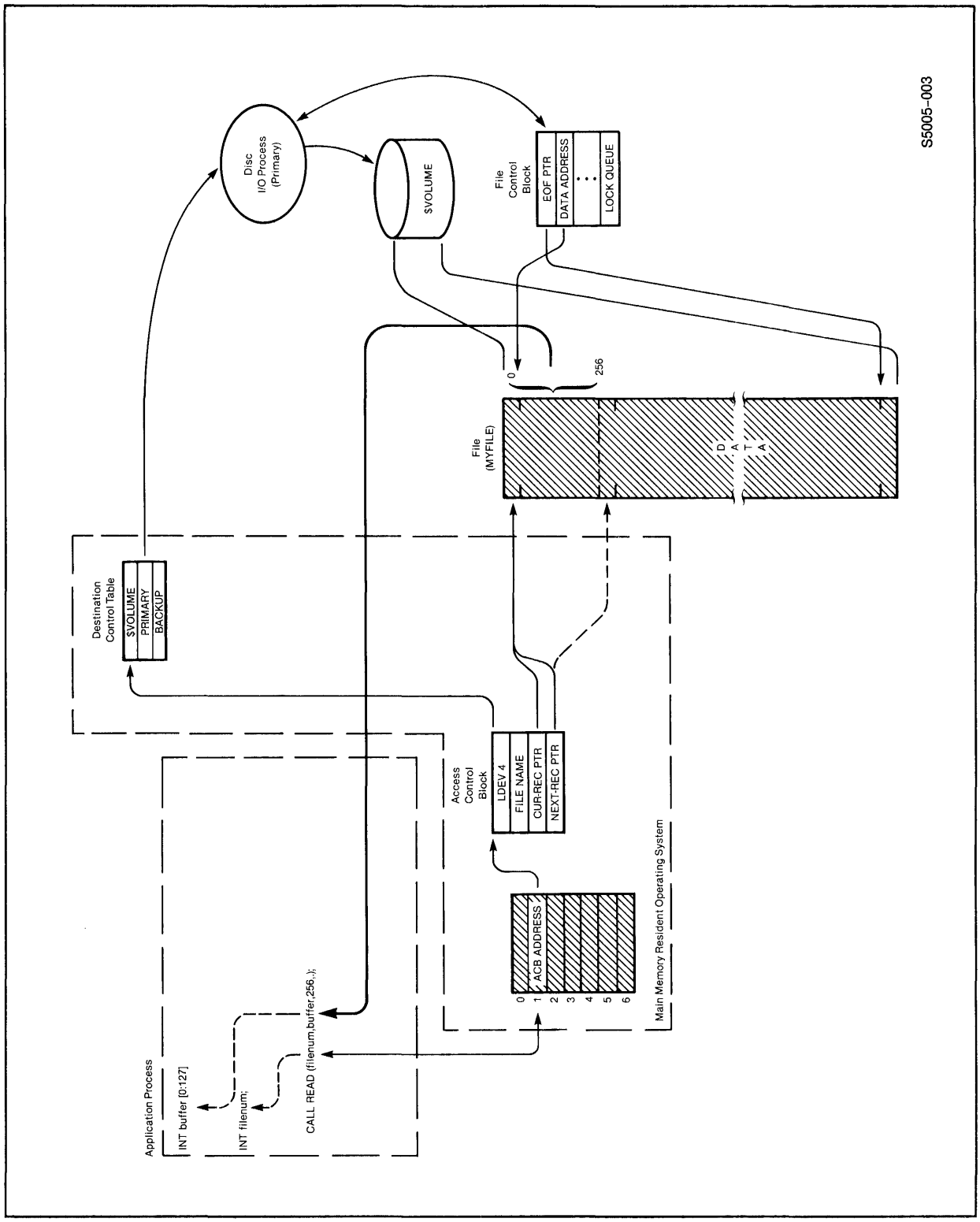


Figure 2-17. File Transfer

Buffering

Two operating system buffers and an application buffer are involved in an I/O transfer. The operating system buffers (a file-system buffer and an I/O buffer) are shown in Figure 2-18.

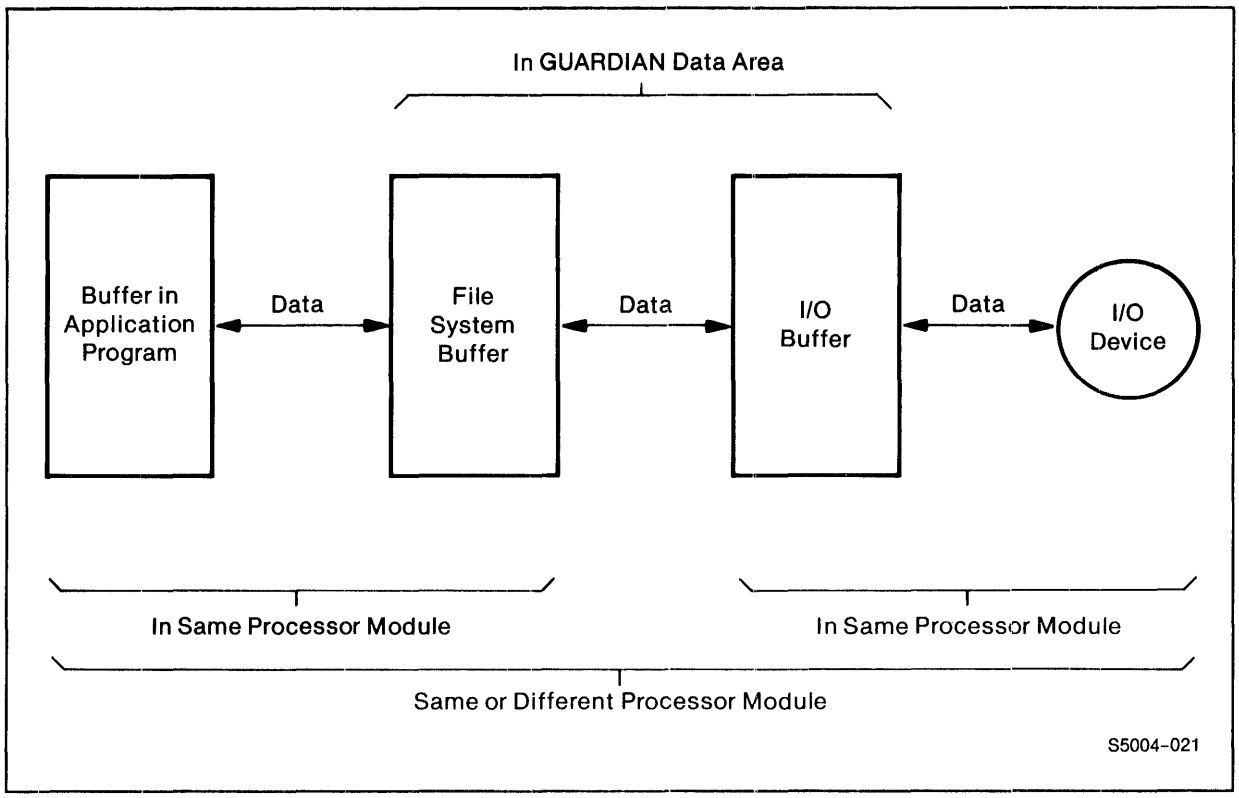


Figure 2-18. Buffering

When an I/O transfer is initiated (in this example a call to READ from a disc file) the file system first secures resident file system buffer space in the processor module where the application process is executing. The amount of file-system buffer space secured is dependent on the transfer count specified in the file system procedure call. Next, the I/O process in the primary processor module controlling the disc is instructed (in this example) to read a block of data from the disc.

The I/O process first secures resident I/O buffer space in its processor module (the amount of I/O buffer space secured is dependent on the transfer count specified in the file-system procedure call), then initiates the I/O transfer. When the I/O transfer with the device is completed, the data is moved from the

I/O buffer in the device's processor module to the file-system buffer in the application's processor module. If these are different processor modules, this is accomplished by an interprocessor bus 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 (virtual) data area.

On NonStop 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.

Closing Files

When a file is closed, the communication path to the file is broken. The ACB is deleted, and the space that it used is returned for use as another ACB. With disc files, if no other opens are outstanding for the file, then the FCB 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 FCB.

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

The condition code should be checked immediately following each call to a file-system procedure. Typically this is done using an IF statement to detect if an error occurred and a call to the file-system procedure FILEINFO to obtain the error number associated with that error, as shown below:

```
CALL READ(FILENUM,BUFFER,72,NUMXFERRED);
IF <> THEN ...
!   if the not equal condition is detected, an error or
!   warning condition has occurred.
  BEGIN
    CALL FILEINFO(FILENUM,ERROR);
        .
        .
  END;
!   returns, in ERROR, the error number associated with
!   the last operation with the file represented by FILENUM.
```

The specific cause of an error or warning condition code is described by an error number. Some error numbers that tend to be observed more frequently than others are mentioned below.

A complete listing of all file-system errors and messages appears in the System Messages Manual.

ERROR RECOVERY

In general, errors can be categorized as follows:

- No error (error number 0, CCE, or operation successful)
- Informational (error numbers 1 through 9, or warnings)
- Soft (recoverable)
- Hard (not recoverable)

- Communication-path errors (recoverable)

Informational errors are those classified as warnings. For example, these include:

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

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:

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

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

Hard errors are those for which programmatic recovery is not possible. These include:

12	file in use
14	device does not exist
43	unable to obtain disc space for extent
45	file is full
48	security violation, illegal or no remote password
49	access violation
150	end of tape detected

Communication-path errors are indicated by error numbers in the range of 200 through 229. A path error is indicated when a failure occurs in the primary path to a device while an I/O operation is in progress. An application process recovers from a path failure by reexecuting the operation (the file system automatically reroutes the I/O request across the alternate path).

Specific file-system errors and error recovery procedures are described in the applicable sections of this manual. A complete listing of all errors appears in the System Messages Manual.

Error message numbers 300 through 511 are reserved for customer use.

Automatic Communication-Path Error Recovery for Disc Files

Operations with disc files are classified as either retryable or nonretryable. Retryable operations are those that can be retried indefinitely, without the possibility of loss or duplication of data. The retryable operations are reading and full-sector writing. Nonretryable operations are those that, if retried, could cause a loss or duplication of data. The nonretryable operations are partial-sector writing and appending to the end of a file.

A sync ID and a requester ID are associated with each distinct file operation and are kept in the file's ACB. The sync ID identifies a single operation in a series of operations; the requester ID identifies the process requesting an I/O operation; together they identify a particular operation requested by a particular process. Also, each disc I/O process maintains a list of completed operations, each operation being identified by a sync ID and a requester ID; these are kept in the FCB.

When an application program calls a file-system procedure to write to disc, the file system initiates an I/O operation by sending an I/O request message to the primary I/O process for that file. The I/O request message contains the data to be written, along with a sync ID, the requester ID, and the address where the data is to be written.

The primary I/O process, upon receipt of the request, stores the information contained in the message and begins processing the request.

If the request involves a nonretryable operation, special action is taken. The primary I/O process first reads the sector to be changed and updates the sector image in memory (when writing a partial sector). The primary I/O process then sends the new or updated sector image in a checkpoint message to its backup I/O process along with the disc address of where it is to be written, the sync ID, and the requester ID. Next, the primary I/O process performs the physical I/O operation to the disc. Upon completion of the I/O operation, the primary I/O process informs the file system (which, in turn, notifies the application process) of the completion.

If the request involves a retryable operation, the information kept by the file system (that contained in an I/O request message) is enough to reinitiate the operation. Therefore, in writing full sectors, no checkpointing occurs between the primary and backup I/O processes.

If a failure of the primary I/O process's processor module occurs, the file system and the backup I/O process are notified.

The backup I/O process, when notified of the primary's failure, takes over the primary's duties. The first action that the backup process 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 process.

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

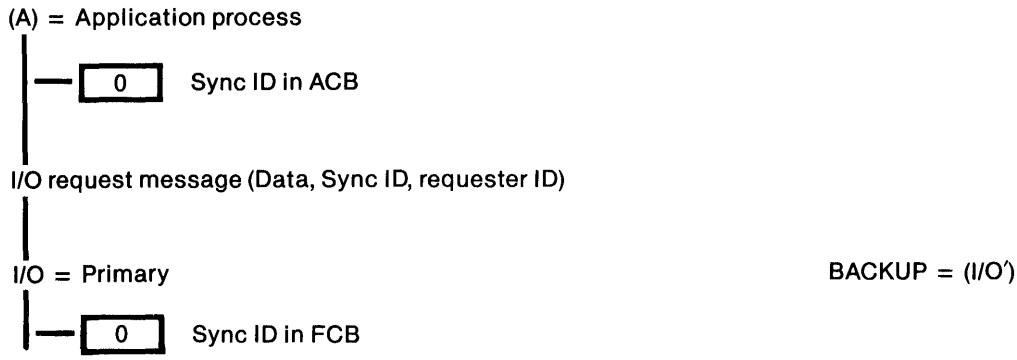
After taking over from its primary process, the backup I/O process checks the sync ID and requester ID in the I/O request message for a match in the list of completed operations. If there is a match, the requested operation has already completed, and the backup I/O process returns the associated completion 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.

This automatic communication-path error recovery can occur only if the file was opened with an explicit <sync-depth> parameter greater than zero. The default <sync-depth> is zero. For more information, refer to the OPEN procedure explanation in the System Procedure Calls Reference Manual.

EXAMPLE 1: Compare this example of a routine I/O operation, performed without incident, with example 2, in which a processor fails.

```
CALL WRITE(FILE^NUMBER,...);
```

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.

S5004-024

*3. The state of the operation about to be performed is checkpointed to backup I/O process. The checkpoint message contains the requester ID, the updated sector image, and the next sync ID.

S5004-025

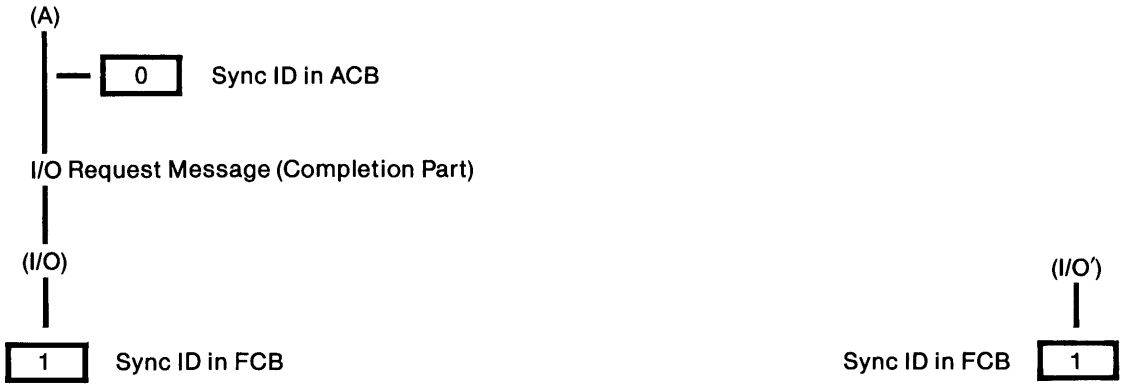
The backup I/O process saves the updated sector image and saves the next sync ID as 1.

4. The primary I/O process then writes the updated sector image to disc.

S5004-026

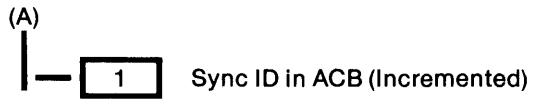
* performed only if a partial sector is to be written

5. The primary I/O process indicates to the application process (through the file system) that the operation is completed.



S5004-027

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

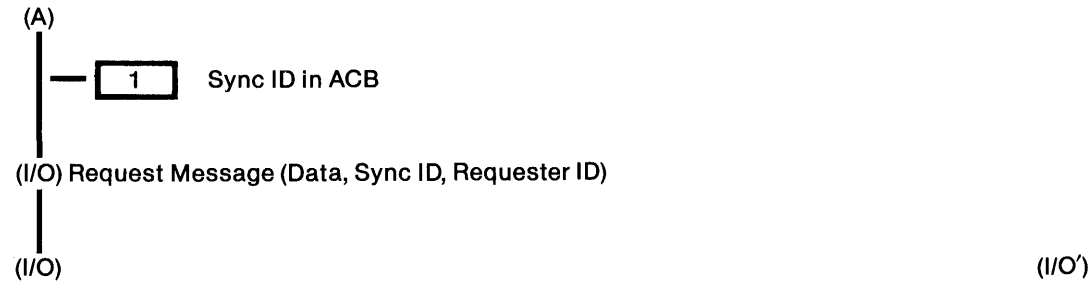


S5004-028

EXAMPLE 2: This example shows an I/O operation in which a processor fails.

```
CALL WRITE (FILE^NUMBER,...);
```

1. The file system sends an I/O request message to the primary disc I/O process.

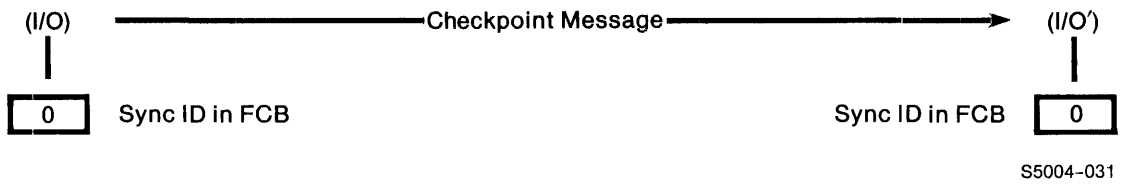


S5004-029

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.

*3. The state of the operation about to be performed is checkpointed to the backup I/O process. The checkpoint contains the requester ID, the updated sector image, and the next sync ID.

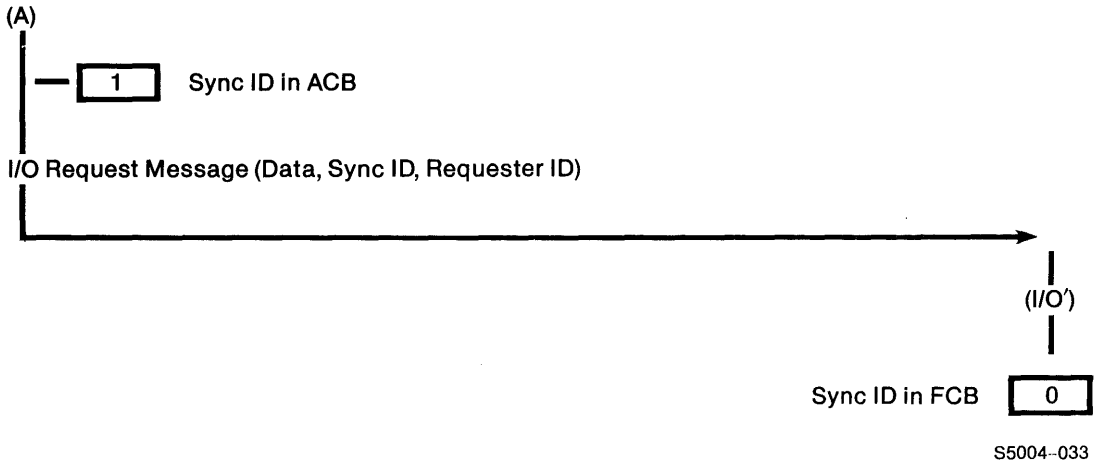


The backup I/O process saves the updated sector image and saves the next sync ID as 0.

4. The primary's processor module fails and the backup I/O process is notified of the failure. (*)The backup I/O process, using the latest checkpoint from the primary, performs the I/O operation to the disc.

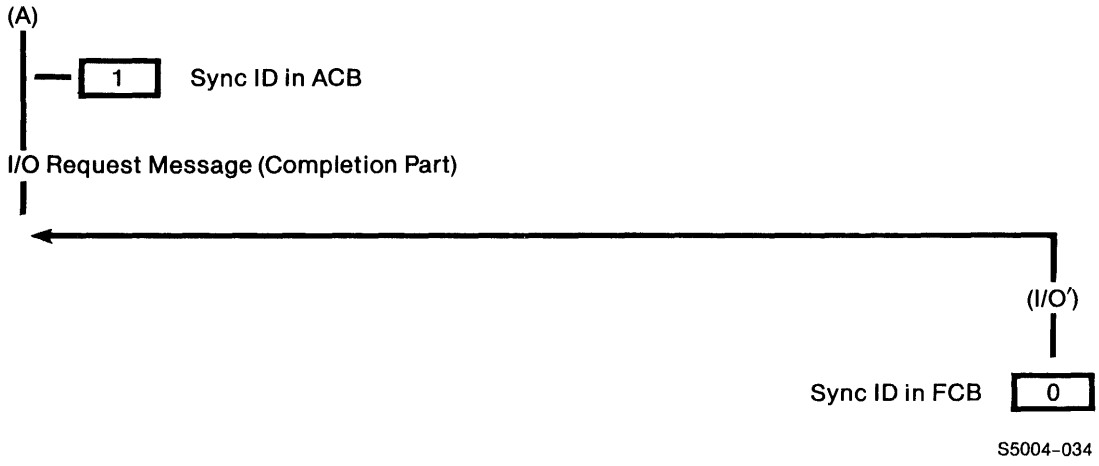
* performed only if a partial sector is to be written

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



S5004-033

6. The backup I/O process compares the requester ID and sync ID in the I/O request message with those of operations that have already been 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.



S5004-034

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

S5004-035

* performed only if a partial sector is to be written

Mirrored Volumes

A mirrored volume is a pair of physically independent disc devices that are configured and accessed as a single volume. One is the primary volume and the other is the mirror volume. Each device is usually controlled through two independent disc controllers. Each mirrored volume is controlled by a separate I/O process pair. The mirroring designation for a volume is selected during system generation. The hardware configuration of a mirrored volume is shown in Figure 2-19.

When writing data to a mirrored volume, the primary I/O process automatically writes the data on both disc devices comprising the volume. As long as both devices are operable, either one can be used by the I/O process for reading because the content of both discs is the same. If one of the devices becomes inoperable, the I/O process performs all subsequent reading from the operable device.

When an inoperable disc device is repaired, the information on the previously inoperable pack is brought up to date by a PUP REVIVE command issued by the operator. REVIVE 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 can be used to specify a time interval between copying groups of tracks. This permits the revive operation to take place without significant degradation of system performance.)

Four options are provided to optimize mirrored volume performance when both devices of a mirrored volume are operable. The reading options, specified at system generation, are SLAVESEEKS or SPLITSEEKS. SPLITSEEKS is the default setting.

- SLAVESEEKS specifies that, when reading, both devices of a mirrored volume are to seek (position the head) together.

- 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.

The writing options, SERIALWRITES or PARALLELWRITES, are selected by the driver--if the controller is shared, SERIALWRITES is used; if not, PARALLELWRITES is used. To obtain the added reliability of SERIALWRITES (one write at a time), SETMODE function 57 can be specified for use by one file.

- SERIALWRITES specifies that the actual data transfer completes on one device before beginning on the other.
- PARALLELWRITES specifies that data transfers to both devices occur concurrently. This option is allowed only if each device is controlled by a separate hardware controller.

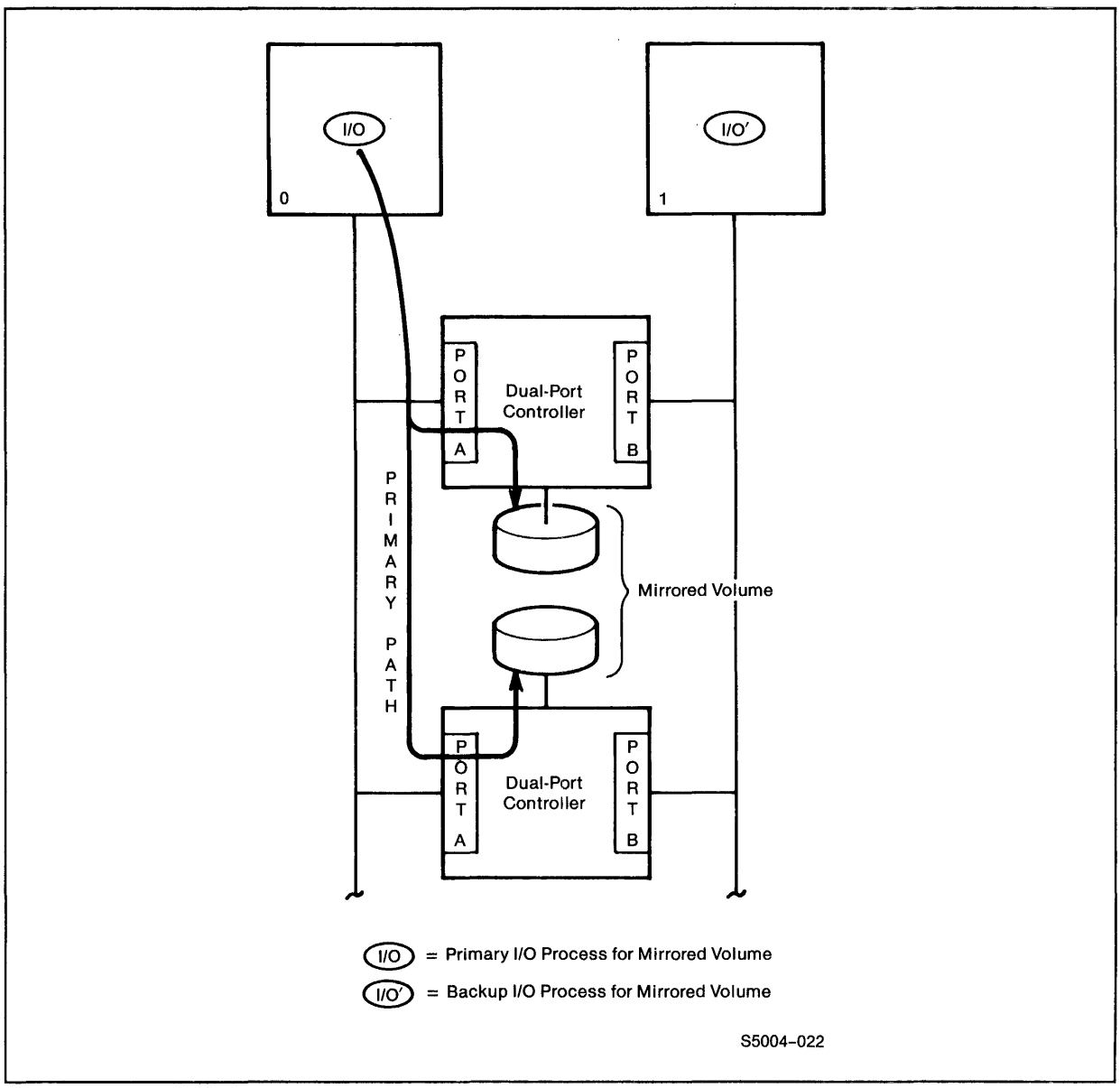


Figure 2-19. Mirrored Volume

SECTION 3

MANAGING PROCESSES

When any program runs on the system, it is called a process. A process is the basic work unit of the GUARDIAN operating system.

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 (whether an application or system program) can be executing concurrently a number of times in the same processor or in different processors; each execution is considered a separate process. This difference is illustrated in Figure 3-1.

A process consists of:

- A code area in virtual memory that contains the instruction codes to be executed (this is shared by all processes in that processor executing the same program file). A process's virtual code area is the code part of the program file on disc.

- A data area in virtual memory that contains the program variables and temporary storage (the memory stack) that is private to the process. (Even if other processes use the same code area, each has its own private data area.) The virtual data area is obtained from the volume where the program file resides.

- A process control block (PCB), identified by a process number, that is used by the operating system to control process execution. The PCB contains pointers to the process code and data areas (real and virtual), retains the current state of the process when the process is suspended, and contains pointers to files opened by the process.

- A process ID that is assigned by the operating system when a program is first called for execution. A process ID consists of three parts:

  1. if a nonnamed process, a timestamp of when the process was created or, if a named process, its symbolic process name,

  2. the number of the processor module where the process is executing, and

  3. the number of the process in that processor.

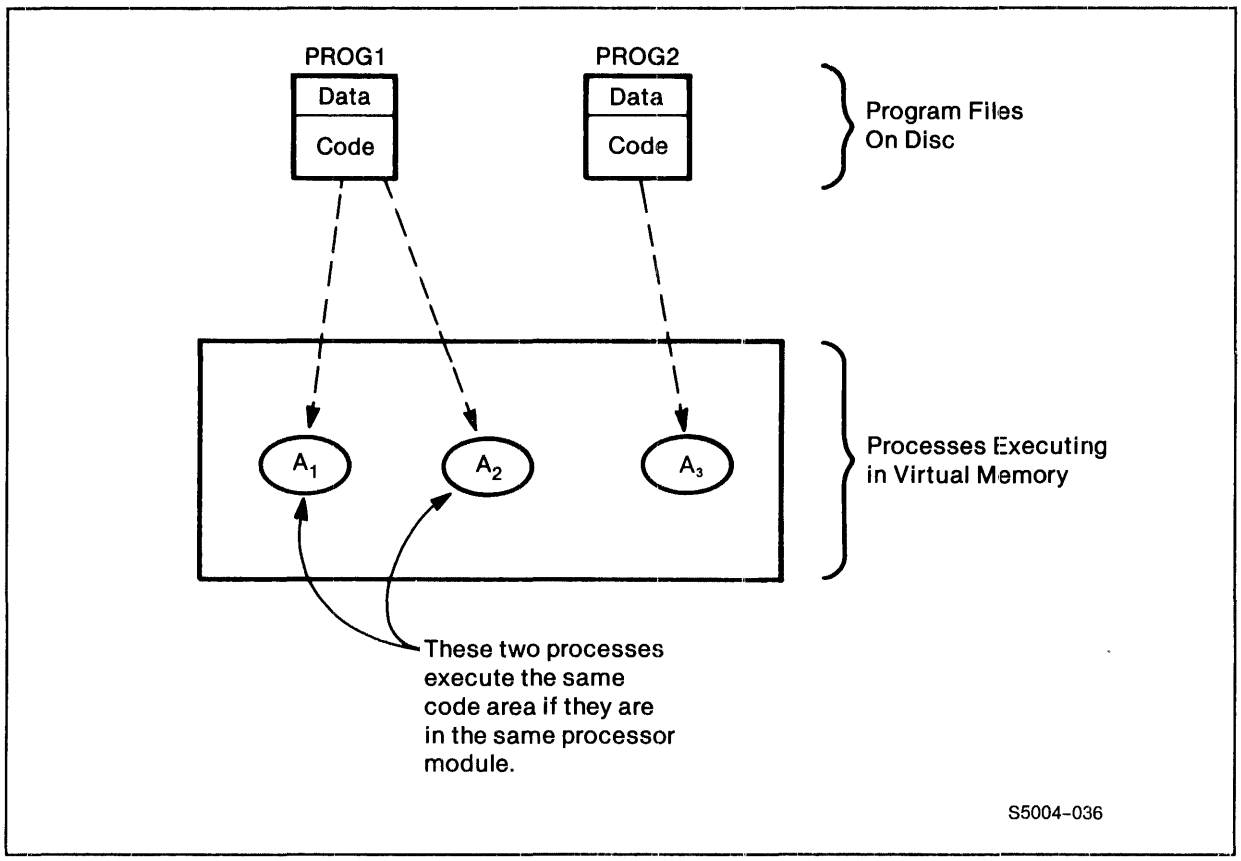


Figure 3-1. Program Versus Process

Figure 3-2 illustrates a process.

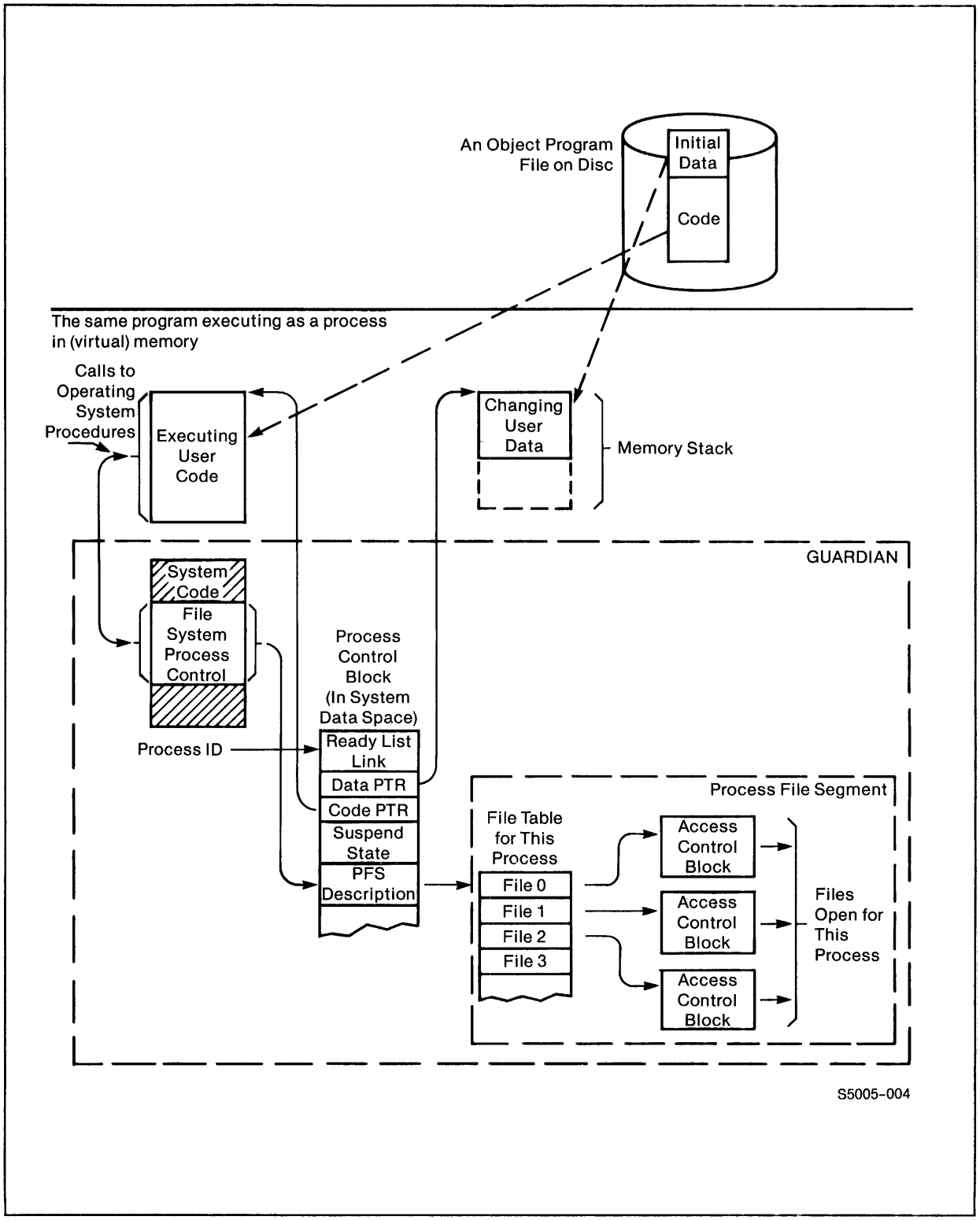


Figure 3-2. A Process

PROCESS STATES

During its existence, a process goes through the following states:

CREATION ⟶ EXECUTION ⟶ DELETION

Process Creation

The term "process 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 and other control blocks
- Determining if the code is being executed by another process (for code sharing)
- Allocating space for copies of the data and (if not sharing code) code segments
- 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.)
- If the process is named, an entry is made for the process in the destination control table (DCT). (See "Named Processes" later in this section.)

Process creation is initiated by use of the process control procedures NEWPROCESS or NEWPROCESSNOWAIT--either interactively through the GUARDIAN command interpreter RUN command or programmatically by application programs. 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. (See the System Procedure Calls Reference Manual for syntax details.)

Process Execution

A process has four executing states:

1. Active--currently executing instructions

2. Waiting--not executing instructions. Instead, the process is suspended, waiting for some external event such as an I/O operation to complete.

3. Ready--executable but not executing instructions. The awaited external event has completed, but another, higher-priority process is currently executing. A process is also in the ready state while waiting for an absent memory page.

4. Suspended--not executing instructions. The process has been removed from the active or ready state and is not waiting. A process is entered into the suspended state when it is the object of a call to the SUSPENDPROCESS procedure. A process is returned to the ready or active state from the suspended state when it is the object of a call to the ACTIVATEPROCESS procedure.

Processes are scheduled for execution according to an application-defined priority assigned when created. The execution priority is determined by a numerical value ranging from 1 (lowest priority) to 255 (highest priority). Processes execute according to their relative priority as follows:

- The ready process with the highest priority executes exclusively until suspended. Process suspension occurs while "wait" input-output operations take place, while waiting for a needed system resource to become available, when a call to the process control DELAY procedure is made, or when the process is the object of a call to the process control SUSPENDPROCESS procedure.

- When more than one process has the same priority, they are executed as follows: the first ready process executes until suspended, then the next ready process executes until suspended, and so on. No preemption occurs among processes having the same priority.

- Ready processes having a lower priority execute only while higher-priority processes are suspended. A higher-priority process becoming ready preempts all lower-priority processes.

Processor time is allocated to processes by an operating system function called the dispatcher. When the operating system determines that a process is ready to execute, the process is placed on the ready list according to the process priority

number. The ready list consists of the PCBs of all ready processes linked together according to their relative priority numbers. When the currently executing process completes or is suspended (for example, while its I/O occurs), or a higher-priority process becomes ready, the dispatcher gives control of the processor to the highest-priority process ready for execution.

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 includes a floating priority feature. If a single process retains uninterrupted control of a CPU for a given period, and other processes of equal or lower priority are thus prevented from running, the operating system will automatically reduce the priority of that process so that other processes may run. Each time the process runs uninterrupted for more than 2 seconds, its priority is reduced by one and timing begins again, reducing the process priority in a stepwise manner. The PRIORITY procedure can be used to check if reduction of priority has occurred.

The following process control procedures are related to process execution:

ACTIVATEPROCESS	returns a process from the suspended state to the ready state.
ALTERPRIORITY	alters 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	detects a looping process. SETLOOPTIMER permits a process to set a limit on the total amount of processor time it is allowed. If the time limit is reached, a process-loop-timer timeout trap occurs.
SUSPENDPROCESS	puts another process into the suspended state.

Process Deletion

Process deletion is the act, by the operating system, of stopping further process execution. The deleted process is removed from its current execution state (active, ready, or suspended), files it has opened are closed, its associated resources (PCB, memory stack space, code space if not shared) are returned to the system, and the deleted process's creator is notified of the deletion by a system message.

There are two types of process deletion, normal and abnormal:

- Normal deletion is initiated by a call to the process control STOP procedure.
- Abnormal deletion is initiated by the call to the process control ABEND procedure or by a trap occurring and certain other conditions being present (see Section 13).

Process deletion can be initiated by a process itself, by another process, or, if a trap occurs, by the operating system.

The following process control procedures relate to process deletion:

STOP	stops a process
ABEND	aborts a process
SETSTOP	controls whether a process can be deleted by any process but itself or its creator (see "Creator" later in this section).

PROCESS ID

A process is uniquely identified by its process ID. There are two equivalent forms of process ID: the timestamp form for nonnamed processes and the process name form for named processes. These forms are the same as those presented in Section 2 under "Process IDs and Process Names".

- Timestamp form

  For nonnamed processes, the GUARDIAN operating system assigns the process ID when the process is created. The general form of this type of process ID is:

```
<process-id>[0].<0:1>   = 2
<process-id>[0].<2:7>   = unused
<process-id>[0].<8:15>  = <system-number> is 0 through 254
<process-id>[1:2]       = low-order 32 bits of creation
                          timestamp
<process-id>[3].<0:3>   = unused
<process-id>[3].<4:7>   = <cpu> where the process is
                          executing
<process-id>[3].<8:15>  = <pin> assigned by operating
                          system to identify the process
                          in the CPU
```

- Process-Name form (Local)

  For named processes, the <process-id> contains an application-defined process name. The local process-name form is:

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

  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 backslash)
<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

<process-name> in words 1 and 2 does not include the initial dollar sign "$".

The following process control procedures relate to process IDs:

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.

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.
- A process can obtain the process ID of its creator by calling the process control MOM procedure.
- The process ID of the originator of the last message received can be obtained by calling the file-system LASTRECEIVE procedure.
- A process name can be predefined and hard coded into the program.
- A process name can be received in a PARAM message from a command interpreter.
- A process can obtain its own <cpu,pin> by calling the process control MYPID procedure.
- A process can obtain the process ID associated with a <cpu,pin> through the process control GETCRTPID or GETREMOTECRTPID procedure.
- A process name can be obtained by a call to CREATEPROCESSNAME, or CREATEREMOTENAME, or PROCESSINFO.
- A process ID is also contained in certain system messages.

The following example illustrates a process ID returned from the process control MOM procedure:

```
INT .FNAME[0:11] := 12 * ["  "];
CALL MOM(FNAME);
```

CREATOR

The term "creator" refers to the relationship that exists between a process that initiated a process creation (that is, 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 (notification is in the form of a Process STOP or Process ABEND system message):

(CI) the command interpreter is notified when (A) is deleted.

This notification is in the form of a system message. Either of two messages may be sent. The messages and their formats, in word elements, are:

- Process normal deletion (STOP) message:

```
<sysmsg>              = - 5
<sysmsg>[1] FOR 4     = process ID of deleted process
```

  This message is received by a deleted process's creator when the deletion is due to a call to the process control STOP procedure.

- Process abnormal deletion (ABEND) message:

```
<sysmsg>              = - 6
<sysmsg>[1] FOR 4     = process ID of deleted process
```

  This message is received by a deleted process's creator when 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.

The system messages are presented in the System Messages Manual.

The process ID of a process's creator is kept in its PCB.

The following procedures are related to a process's creator:

MOM provides a process with the process ID of its creator.

SETSTOP protects a process from being deleted by any process but itself or its creator.

STEPMOM permits a process to assume the role of creator of an already created process and, therefore, receive its process deletion messages.

PROCESS PAIRS

For the purpose of implementing fault-tolerant applications, the concept of the process pair is used. Typically, but not necessarily, a process pair is two related executions of the same program code in separate processor modules for the purpose of providing fault-tolerant operation. (See Figure 3-3.)

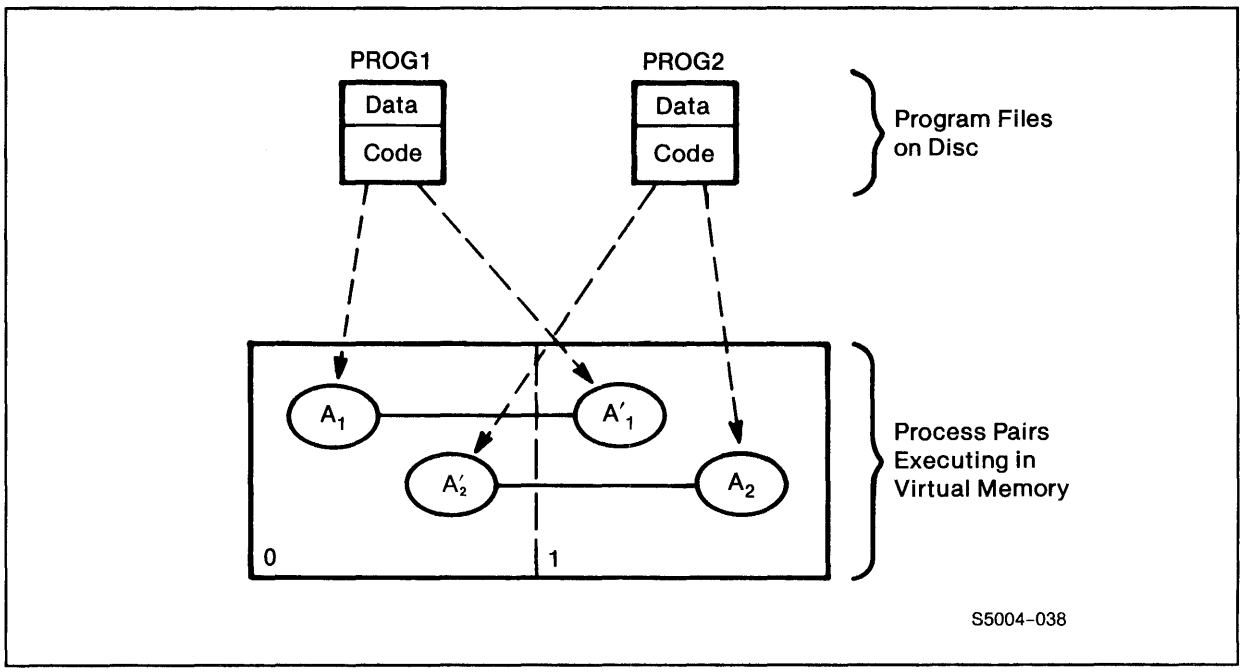
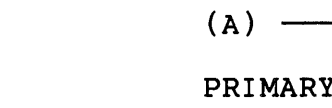


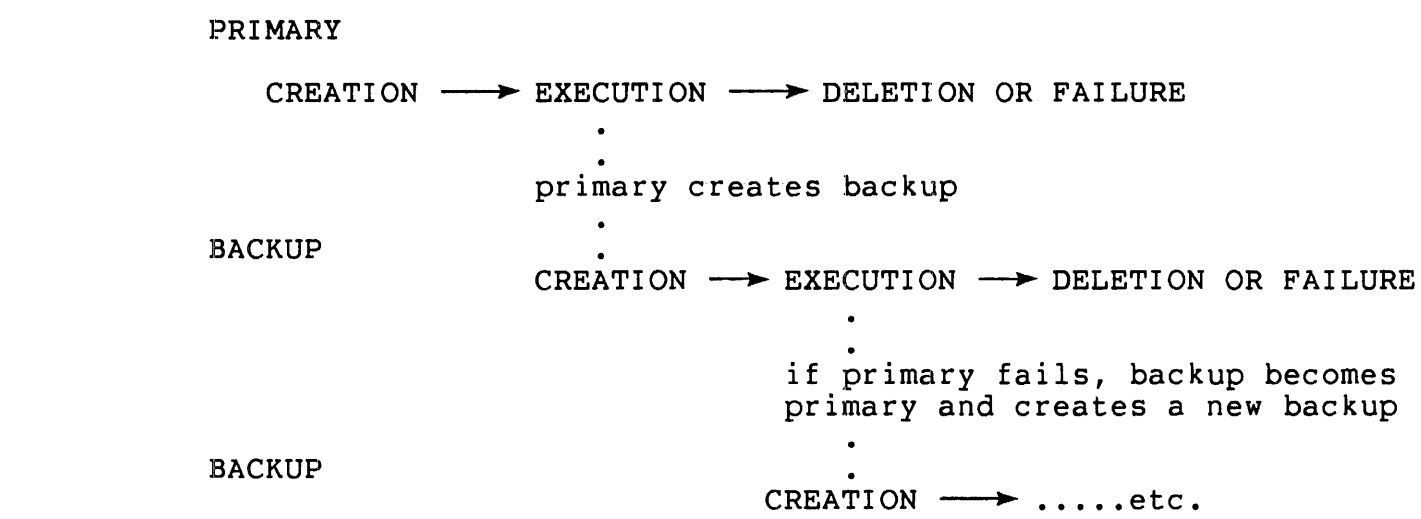
Figure 3-3. Process Pairs

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.



During a process pair's existence, it passes through the following process execution states:



For a complete explanation of the individual duties of each member of a pair, refer to Section 12.

NAMED PROCESSES

Named processes provide four major benefits:

1. A pair of processes is treated as a single entity.

2. Each member of a pair is related to the other so that one member is notified when the other is deleted.

3. A process or process pair is related to a process, called an "ancestor process", to be notified when the pair (as an entity) no longer exists.

4. A process or process pair can be known by a predefined symbolic file name, so that other processes can easily communicate with the pair by that name.

A process pair is identified by its process name in a directory called the PPD. One or two processes may be associated with a given name. The PPD is a part of the destination control table.

(To view the PPD, enter the letters "PPD" in response to a COMINT prompt.)

Operation of the PPD

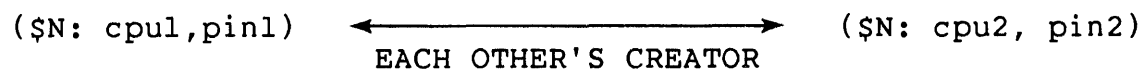
Each entry in the PPD, in words, is of the form:

```
          word
         [0:2]          [3]       [4]       [5:8]
  entry#
   [0]  $<process-name><cpu,pin1><cpu,pin2><ancestor-process-id>
   [1]          .          .         .              .
    .           .          .         .              .
    .           .          .         .              .
    .           .          .         .              .
   [n-1]        .          .         .              .
```

Each entry consists of a process name, the <cpu,pin>s of the two processes that make up 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 (part of the DCT) at process creation time when the <name> parameter is included in a call 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 (backup) process associated with its name. A process can have the system generate, by 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 besides the creator, but the fault-tolerant aspects of named processes are desired. (A process name, either predefined or system-generated, can be assigned by the GUARDIAN command interpreter RUN command; see the GUARDIAN Operating System User's Guide.)

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 = process name of a named process pair)

When a process represented in the PPD by a given name is deleted or its processor module fails, the reference to the particular process (<cpu,pin>) is zeroed in the PPD, and the other process

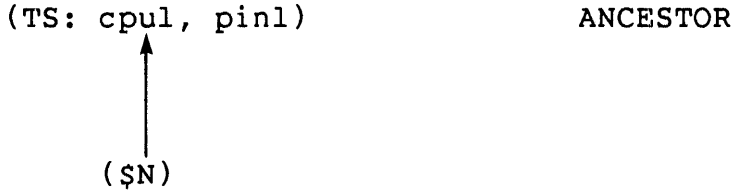
(if any) becomes the primary. When the new primary creates a new process having that name, the new process's <cpu,pin> is entered into the PPD.

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

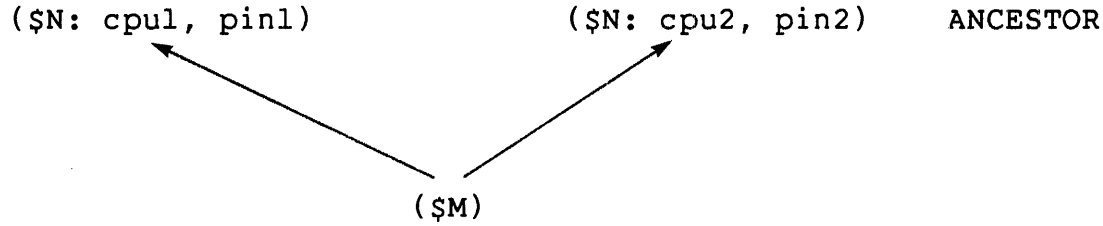
The ancestor relationship can exist between the following:

- A nonnamed process and a named process pair



  where the single nonnamed process (TS: cpu1, pin1) is notified when the last process having the name ($N) is deleted. (TS = timestamp of a single nonnamed process)

- Two named process pairs



  where both members of the named process pair ($N) are notified when the last process having the name ($M) is deleted.

The notification to the ancestor process is in the form of a system message such as processor failure, process STOP, or process ABEND.

- CPU down message:

```
<sysmsg>              = -2
<sysmsg>[1] FOR 3     = $<process-name>
<sysmsg>[4]           = -1
```

  This message is received by an ancestor process when the indicated process name is deleted from the PPD because of a processor module failure. This means that the named process or process pair no longer exists.

- Process normal deletion (STOP) message:

```
<sysmsg>              = -5
<sysmsg>[1] FOR 3     = $<process-name> of deleted process
                        or process pair
<sysmsg>[4]           = -1
```

  This message is received if a process deletion is due to a call to the process control STOP procedure. 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.

- Process abnormal deletion (ABEND) message:

```
<sysmsg>              = -6
<sysmsg>[1] FOR 3     = $<process-name> of deleted process
                        or 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.

The system messages are presented in the System Messages Manual.

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.

Reserved Process Names

The following names should be avoided when choosing process names. The names listed here are used by Tandem software to refer to specific system processes or devices. Use these names only to refer to the appropriate processes or devices.

```
$AOPR
$CMON
$CMP
$C9341
$DM<nn>
$IMON
$IPB
$MLOK
$NCP
$NULL
$OSP
$PM
$S
$SPLS
$SSCP
$T
$TICS
$TMP
$X<nam>
$Y<nam>
$Z<nam>
```

<nn> is any two digits (00 through 99)
<nam> is any combination of 1 through 3 letters or digits (A through Z, 0 through 9)

The following names are not reserved but should be used with caution because they are commonly used for a specific purpose:

```
$DISC
$LP
$SPLP
$TAPE
```

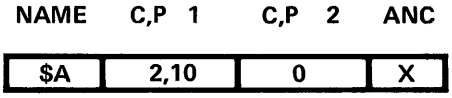
Example Operation of the PPD

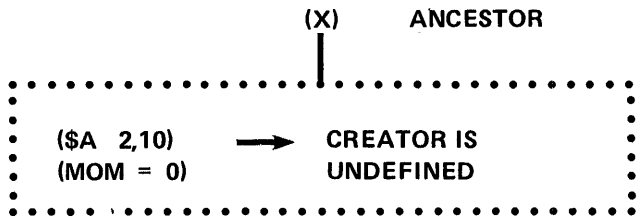
PROCESSES PPD

1. The process (x) exists:

2. (x) initiates creation of a named process pair $A:

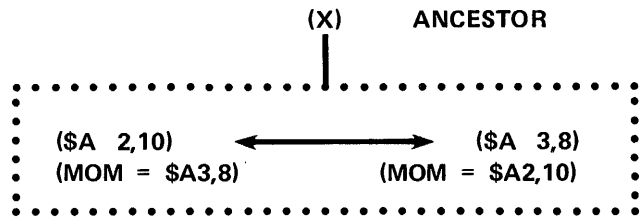
```
name ':=' "$A     ";
CALL NEWPROCESS(...,name);
```





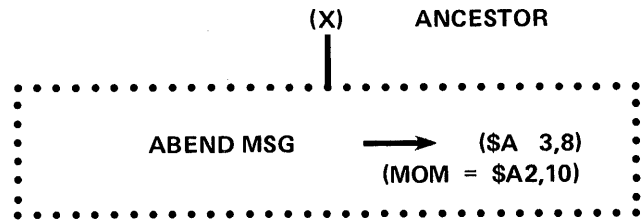
3. ($A 2,10) creates its backup:

```
name ':=' "$A     ";
CALL NEWPROCESS(...,name);
```



EACH OTHER'S CREATOR

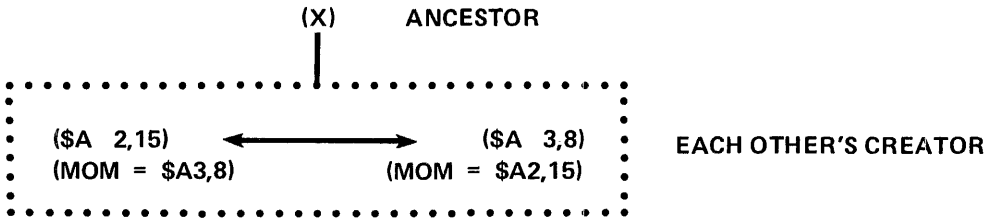
4. ($A 2,10) ABENDs:



5. ($A 3,8) creates its backup:

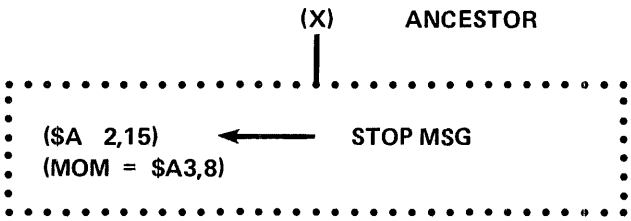
```
name ':=' "$A     ";
CALL NEWPROCESS(...,name);
```

$A	3,8	2,15	X



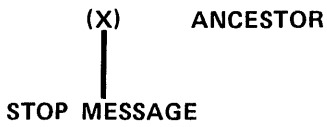
6. ($A 3,8) stops:

$A	2,15	0	X



7. ($A 2,15) stops:

DELETED



Procedures

The following procedures are used for performing operations involving named processes or process pairs:

CREATEPROCESSNAME	generates 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 and, optionally, enters its application-defined symbolic process name into the PPD.

NEWPROCESSNOWAIT	creates a new process in a nowait manner and, optionally, enters its application-defined symbolic process name into the PPD.
STOP	deletes a named process or process pair.

HOME TERMINAL

Associated with each process is its home terminal (figure 3-4). A new process's home terminal may be specified by the optional TERM parameter of the command interpreter RUN command. If the TERM parameter is omitted, which is the usual case, the command interpreter's home terminal is used for the new process's home terminal. The home terminal designation is passed on to descendent processes at process creation time.

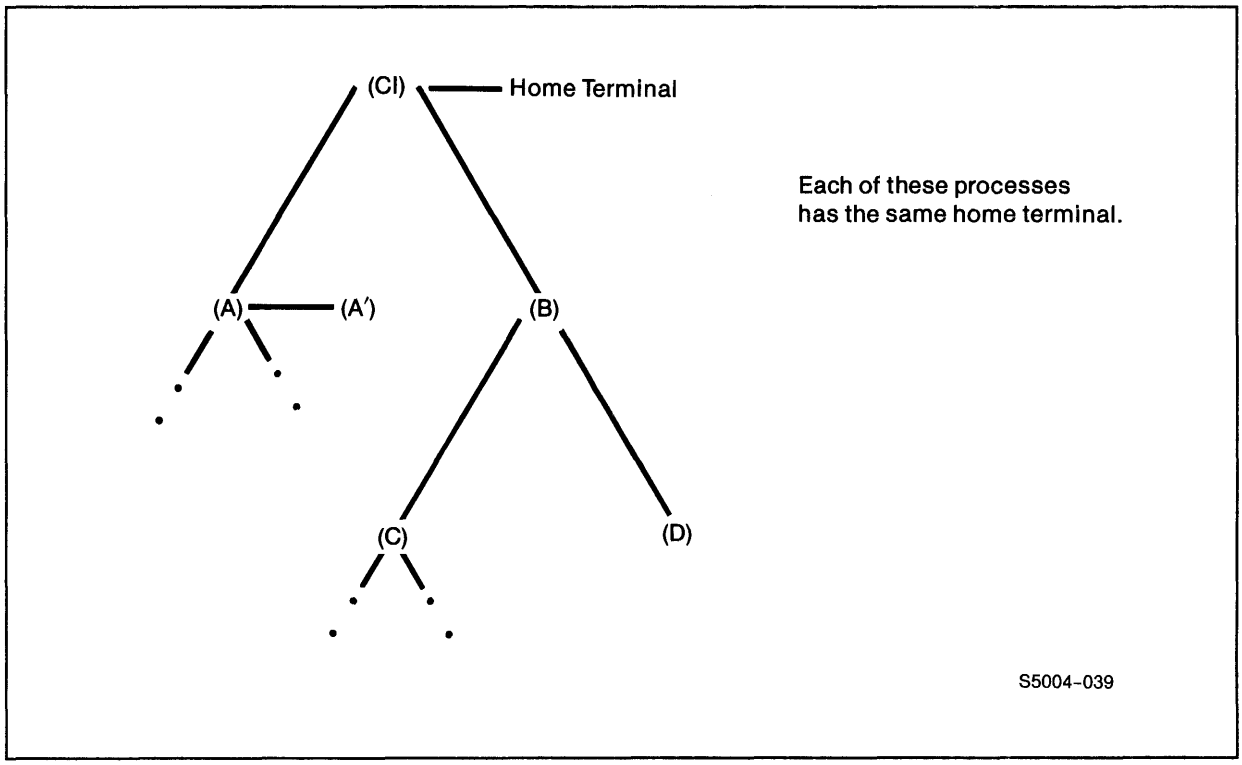


Figure 3-4. Home Terminal

The following procedures are related to a process's home terminal:

MYTERM returns the file name of a process's home terminal.

SETMYTERM specifies a new home terminal device.

PROCESS TIMING

The NonStop system allows a process to set timers that count actual elapsed (real) time or process time (CPU time used only by the process). When the time interval for a timer has expired, the user process receives the timeout as a system message (file-system error 6, CCG).

The following procedures are related to process timing:

SIGNALTIMEOUT sets a timer for a given period of real time.

CANCELTIMEOUT cancels a timer previously set by SIGNALTIMEOUT.

SIGNALPROCESSTIMEOUT sets a timer for a given period of time; similar to SIGNALTIMEOUT but based on process time instead of real time.

CANCELPROCESSTIMEOUT cancels a timer set by a call to SIGNALPROCESSTIMEOUT.

CPUTIMES returns the time spent since coldload in: CPU process busy, CPU interrupt, and CPU idle.

The following procedures pertain to process time as opposed to real (wall-clock) time. They are important in gathering statistics about the CPU execution time a process requires. Process time is defined as the CPU busy time needed to execute the process. It excludes the CPU time to execute software that processes any interrupts that occur while the process executes.

MYPROCESSTIME returns, in microseconds, the process execution time of the calling process.

PROCESSTIME returns, in microseconds, the process time of any process in the network.

CONVERTPROCESSTIME converts the time from microseconds to hours, minutes, seconds, milliseconds and microseconds.

NOTE

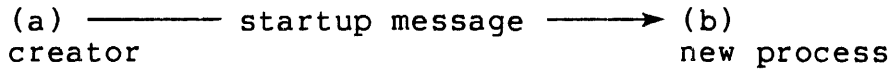
The process time returned by any procedure reporting the time required to execute a process can vary a small amount due to external influences. The differences in process time can be attributed to time spent by the microcode in processing interrupts.

The syntax and considerations for using these procedures is presented in the System Procedure Calls Reference Manual.

CREATING AND COMMUNICATING WITH A NEW PROCESS

The following example shows the use of the NEWPROCESS procedure to run a program and the use of file-system procedures to send and receive a startup message. The example shows the creation of a single nonnamed process. For an example of how to create a named process pair and the action taken by each member of the pair, see Section 12. See Appendix B for a lengthy 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:



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>.

A file is then opened to the new process using the file-system OPEN procedure:

```
    .
CALL OPEN ( PID, FNUM );
IF < THEN ....; ! open failed.
    .
```

Then a message is sent to the new process:

```
    .
BUFFER ':=' "GET TO WORK";
CALL WRITE ( FNUM, BUFFER, 19 );
    .
```

The new process picks up this message by opening, then reading, its $RECEIVE file. Additionally, the new process ensures that the message is from its creator by comparing the process ID returned from the MOM procedure with that returned from the LASTRECEIVE procedure. If the two process IDs do not match, the message is ignored, and the new process reads the $RECEIVE file again:

new process

```
  .
INT .RECEIVE[0:11] := ["$RECEIVE", 8 * ["  "]];
    .PID[0:3], .last^pid[0:3],
    RECV^FNUM, .BUFFER[0:99], NUM^READ;
  .
```

First, obtains the process ID of the creator.

```
  .
CALL MOM ( PID );
  .
```

Then the new process opens and reads $RECEIVE:

```
  .
CALL OPEN ( RECEIVE, RECV^FNUM );
IF < THEN CALL ABEND; ! open failed.
  .
  .
re^read:
  .
CALL READ ( RECV^FNUM, BUFFER, 100, NUM^READ );
IF < THEN ... ; ! error occurred.
  .
```

If the read was successful and the message was from the creator, "GET TO WORK" is returned in BUFFER and 19 is returned in NUM^READ.

The new process verifies that the message is, in fact, from its creator:

```
  .
CALL LASTRECEIVE ( LAST^PID );
IF < THEN ... ; ! error occurred.
  .
IF LAST^PID <> PID FOR 4 THEN GOTO re^read; ! ignore message.
```

The following process control procedures are useful when creating or communicating with new processes:

CONVERTPROCESSNAME	converts a process name from local to network form.
CREATEREMOTENAME	generates a process name for a remote system.
GETPPDENTRY	returns the PPD entry in a remote system associated with an entry number (a number specifying an ordinal position in the PPD).
MYSYSTEMNUMBER	provides a process with the system number (if any) of the system in which it is running.
PROCESSINFO	returns requested information regarding a process's status.
PROGRAMFILENAME	provides a process with the name of its program file.

EXECUTION PRIORITY

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 permits 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 nowait 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 is an overview of suggested priority values for system and user processes. System priorities are set at system generation.

System Processes	Priority	User Processes
Disc I/O processes	220 . .	only processes that do not use virtual memory
Memory Manager, Operator Process, $NCP, etc.	210 210 . .	
System Monitor	200	
Nondisc I/O processes	199	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

The example in Figure 3-5 shows how processor time is divided among three processes executing in the same processor module. Two of the processes are scheduled with an execution priority of 199; the other has a lower priority of 150.

Notice that processor time alternates between the two processes executing with a priority of 199. When one process is suspended for I/O, the other process runs.

The only time that the process with a priority of 150 executes is when both the other processes are suspended. Additionally, the lower-priority process is immediately suspended when a higher-priority process becomes ready.

This example does not account for the effects of system processes; nor does it illustrate floating priorities, which apply only when a process becomes excessively CPU-bound.

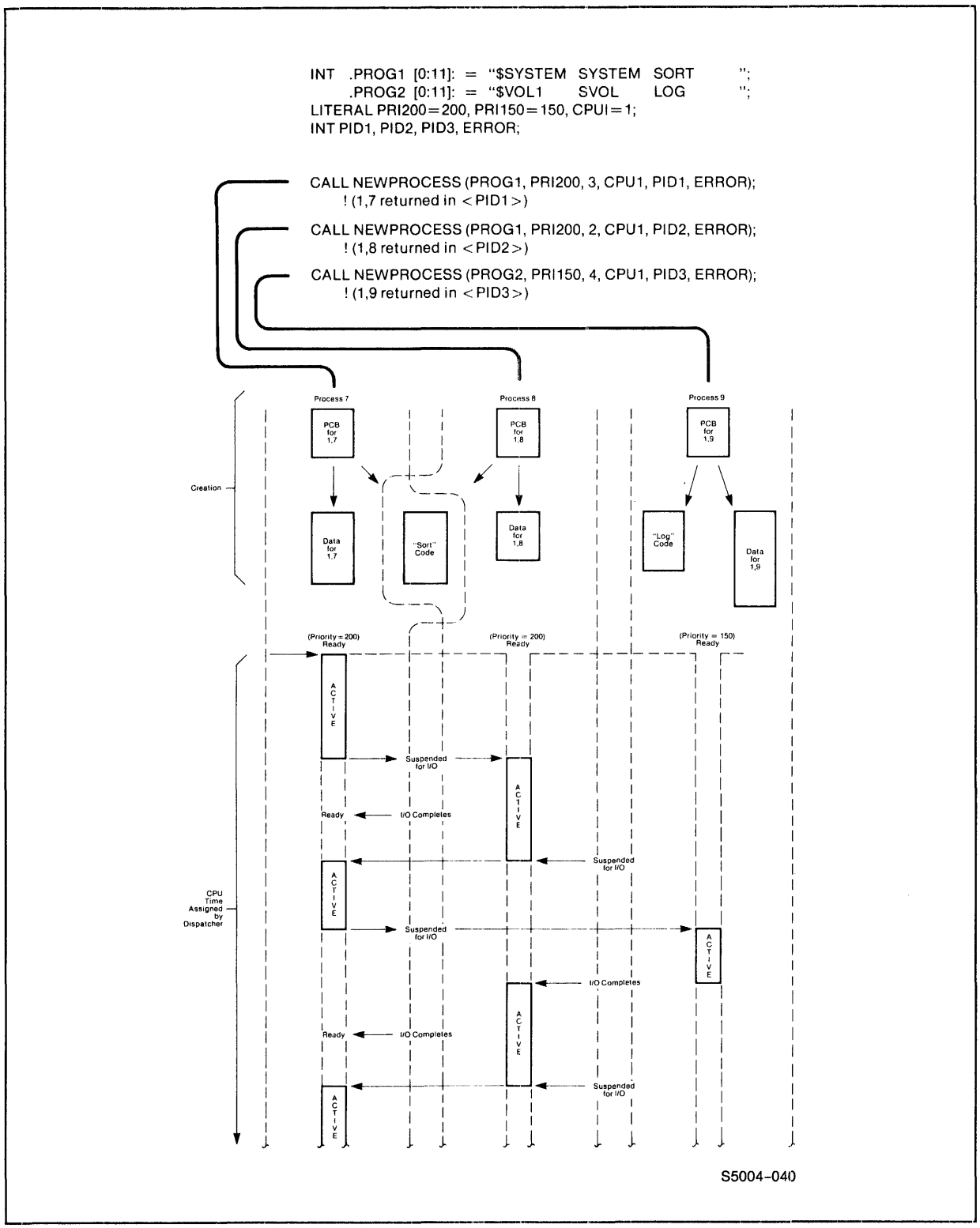


Figure 3-5. Execution Priority Example

SECTION 4

COMMUNICATING WITH OTHER PROCESSES

This section describes the system capability to pass information (such as messages of completion) from one process to another. When more than one process exists, it is often important that a degree of communication can be established between them. The types of communication available, the procedures used to set it up, and the controls needed to avoid potential error are described here.

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 by standard file system procedure calls.

The programming described in this section requires that the processes exist; and to exist they must first be created. For the definition of a process and a description of how processes are created and controlled, see Section 3.

The example given with "Creating and Communicating with a New Process" near the end of Section 3 shows how to use the NEWPROCESS procedure to run a program and how to use the file system procedures to send and receive a startup message. The startup message is explained in Section 5.

A simplified method of starting a process is available to users of sequential files through use of the INITIALIZER. It is described in the System Procedure Calls Reference Manual. Examples of the INITIALIZER procedure are presented in Section 17, "Sequential I/O Procedures".

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 \<sysnum><process-name>

If a network ID is used, <sysnum> is the system number.

The process-name form of the process ID can be further qualified at file open time by adding one or two optional qualifier names. This provides for process file names of the form:

Words:	Content:
[0:3]	$<process-name> or \<sysnum><process-name>
[4:7]	#<1st-qualif-name>
[8:11]	<2nd-qualif-name>

The device type for a file representing a process is 0.

The successful completion of a write to another process means that the process is running and has read the message through its $RECEIVE file.

The WRITE or WRITEREAD <count-written> parameter, when writing to another process, indicates the number of bytes read by the destination process. The <count-read> parameter, when WRITEREAD is called to communicate with another process, indicates the number of bytes returned from the process.

The file system automatically tries all paths when writing to another process if the process was opened with a sync depth greater than zero. An error return of a path error in this case then indicates that the process is no longer accessible.

A "sync ID" scheme is provided that allows a process reading $RECEIVE to detect duplicate requests from requester processes (such duplicate requests are caused by a backup requester process reexecuting the latest request of a failed primary requester process, or by the file system resending the latest request from a requester process because the primary server process failed).

A sync ID is a double-word, unsigned integer whose value is sent along with each interprocess message. Each opener of the process has its own sync ID. Sync IDs 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.

For a process reading the $RECEIVE file, information about the latest message read from the file can be obtained by calling the LASTRECEIVE or RECEIVEINFO procedure. This information includes:

- The process ID of the process that sent the message: This parameter is returned by both LASTRECEIVE and RECEIVEINFO.

- The message tag of the message (see "Types of Communication between Processes" in this section): This parameter is returned by both LASTRECEIVE and RECEIVEINFO.

- The sync ID of the message (see "Sync ID for Duplicate Request Detection" in this section): This parameter is returned by RECEIVEINFO.

- 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 <filenum> 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 (<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> equals 0, a WRITE request or WRITEREAD request with a read count of 0 was made; if <read-count> is greater than 0, then the requester performed a WRITEREAD request of <read-count> bytes. This information can be used to determine if the sender is simply sending data (if <read-count> = 0, then sender is listing); or expects a reply (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 through 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 12.

If a process is coded 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.

SUMMARY OF APPLICABLE PROCEDURES

Use the following procedures to perform input-output operations with other processes:

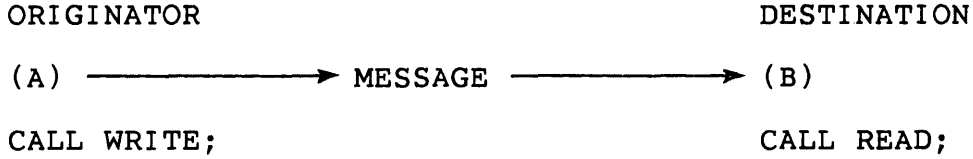
DEVICEINFO	provides the device type and record length for an interprocess file.
OPEN	establishes communication with a file.
READ	reads information from the $RECEIVE file.
READUPDATE	reads a message from $RECEIVE in anticipation of replying to the originator of the message in a subsequent call to REPLY.
LASTRECEIVE	returns the process ID and/or the message tag associated with the last message taken from the $RECEIVE file.
RECEIVEINFO	returns process ID, message tag, error recovery (sync ID) and/or request-related (file number and read count) information associated with the last message read from the $RECEIVE file.
REPLY	replies, by way of the $RECEIVE file, to the originator of a message that was previously received in a call to READUPDATE. Optionally, REPLY uses the message tag returned from LASTRECEIVE to designate which message is to be replied to.

WRITE	sends a message to a designated process referenced by a process ID.
WRITEREAD	sends a message to a designated process referenced by a process ID, then wait for a reply message back from that process.
CONTROL	issues CONTROL operations to a process.
CONTROLBUF	issues CONTROLBUF operations to a process.
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	issues SETMODE functions to a process.
SETMODENOWAIT	performs the same functions as SETMODE, except in a nowait manner on an open file.
CLOSE	stops access to an open file.

TYPES OF COMMUNICATION BETWEEN PROCESSES

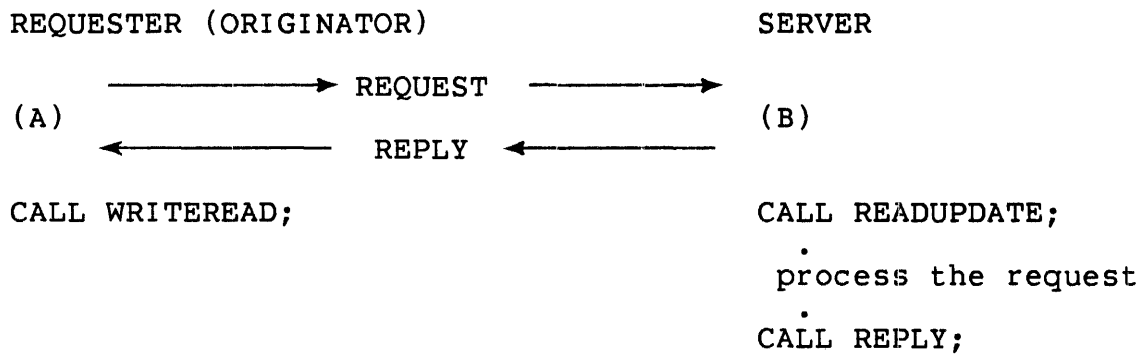
There are two types of communication possible between processes:

- One-way communication (destination calls READ)



  The destination picks up a message by calling the READ procedure. The originator's WRITE completes when the destination's READ completes. If the originator sends a message by 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 (destination calls READUPDATE and REPLY)



The server process picks up a message by calling the READUPDATE procedure then, subsequently, replies to the message in a call to the REPLY procedure. The requester process sends the message and waits for the reply by calling the WRITEREAD procedure. The WRITEREAD completes when the server's REPLY completes. If the requester sends a message by calling the WRITE procedure, the WRITE completes when the destination's REPLY completes. No data is returned to the requester.

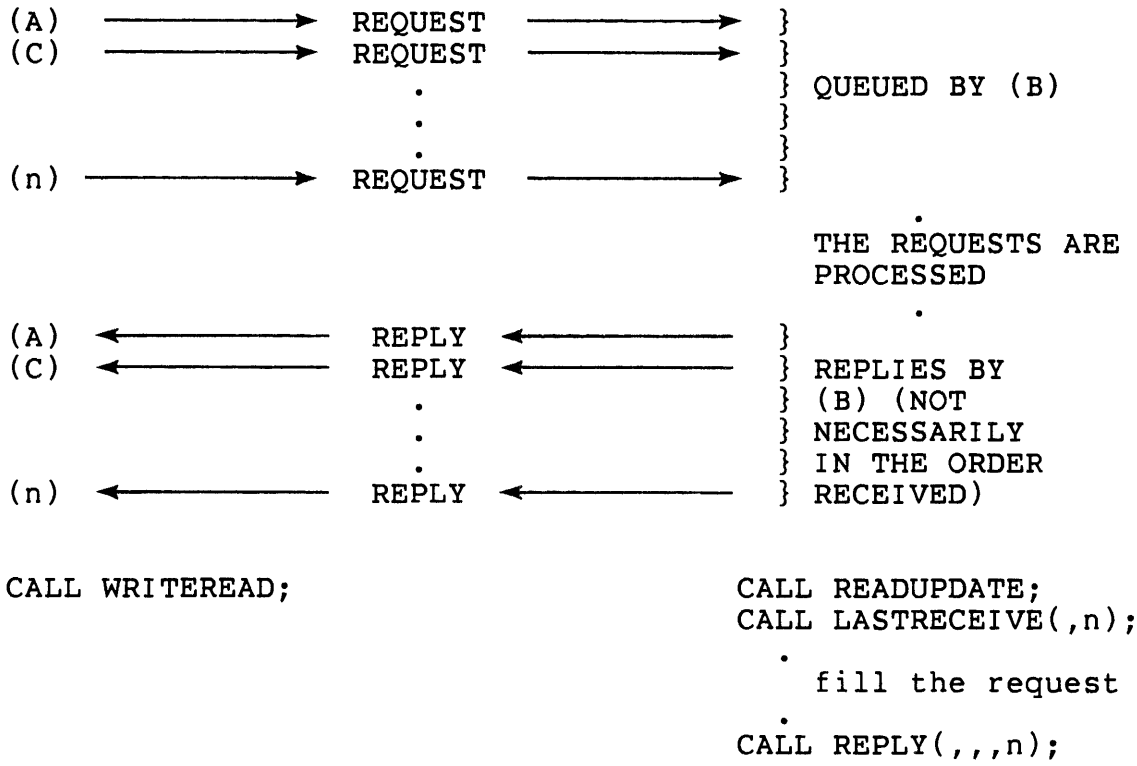
Synchronization

It is important to note that, while a message transfer is in progress, whether it is a one-way or two-way message, the two processes involved in the transfer are synchronized. For example, if there are two processes communicating--processes A and B:

- If process B reads a message by using the READ procedure, the process A call to WRITE or WRITEREAD does not complete (and therefore A may become suspended) until the process B call to READ completes.

- If process B reads a message using the READUPDATE procedure, the process A call to WRITE or WRITEREAD does not complete (and therefore A may become suspended) until the process B call to REPLY completes.

  It is also possible for the server to queue requests before replying:



The maximum number of messages that the server process expects to queue is specified when it opens its $RECEIVE file. To identify each incoming message and direct a reply back to the requester, a message tag is obtained by calling the LASTRECEIVE or RECEIVEINFO procedure immediately following each call to READUPDATE. To indicate which message a response is being made for, the message tag associated with the particular message is passed back to the system when the REPLY procedure is called to make the response. (The message tag is shown as parameter "n" in the above example.)

$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 files, however, reading $RECEIVE does not solicit information from some input device. Instead, reading $RECEIVE reads unsolicited messages that have been sent to a process by way of its process ID or process name.

The following should be taken into consideration when opening the $RECEIVE file:

1. Is nowait I/O desired?

2. Are OPEN, CLOSE, CONTROL, SETMODE, RESETSYNC, and CONTROLBUF system messages desired?

3. 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?

Nowait I/O

If only the startup 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 (the default) specified.

However, if the process must execute concurrently with the receipt of messages, the $RECEIVE file must be opened with nowait I/O specified. If nowait 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 (for example, by setting <filenum> = -1), then an incoming message can be received while waiting for some other nowait I/O operation to complete.

The following illustrates these principles of nowait I/O:

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

CALL OPEN ( RECV^FNAME, RECV^FNUM, 1 ); ! nowait I/O
  .
  .
  .
CALL READ ( RECV^FNUM, RECV^BUFFER, COUNT );
```

At some later point in the program, an operation is initiated for some other file:

```
CALL WRITE ( DISC^FNUM, BUFFER1, ... );
```

Then a call to AWAITIO is made to complete the operation (and also check for incoming messages):

```
WAIT:
   FNUM := -1;  ! wait on any file
   CALL AWAITIO ( FNUM, BUFFER, NUM^READ,, -1D );
```

When AWAITIO completes, the file number returned is compared with the file number of the $RECEIVE file:

```
IF FNUM = RECV^FNUM THEN .....
```

If the file numbers match, a message was received in the $RECEIVE file. The message is processed and, so that another incoming message can be received, another read on $RECEIVE is initiated. Because the call to AWAITIO was actually made to wait on another I/O operation, the program calls AWAITIO again.

```
.
CALL READ ( RECV^FNUM, RECV^BUFFER, COUNT );
GOTO wait;
.
```

the program then returns to the call to AWAITIO to wait for the operation to DISC^FNUM to complete (or for another message to show up on $RECEIVE).

The first message (or series of messages) that a process created by a command interpreter should expect is the startup 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 5, "Interface to the GUARDIAN Command Interpreter" for the message format.

System Messages

If the <flags>.<1> parameter of OPEN is a 1, then a process can receive OPEN, CLOSE, CONTROL, SETMODE, RESETSYNC, and CONTROLBUF system messages through the $RECEIVE file.

These messages are usually desired only if the receiving process is serving several processes or the process is simulating an I/O device. The OPEN and CLOSE system messages permit a process to keep track of its accessors. The CONTROL, SETMODE, and CONTROLBUF system messages permit a process to act as though it were an I/O device. The RESETSYNC system message (see RESETSYNC procedure in Section 12) informs a process when a file's sync ID has been reset.

Communication Type

The value of the <receive-depth> parameter of the OPEN call determines whether or not a process is to perform two-way communication through the $RECEIVE file.

If <receive-depth> is 0, one-way communication is indicated. The receiver can only accept incoming messages; replies cannot be issued. Incoming messages must be read by calls to the READ procedure. Calls to READUPDATE and REPLY are not permitted.

If <receive-depth> is 1 or more, two-way communication is indicated. The receiver can accept and reply to incoming messages. Messages are read by either the READUPDATE or the READ procedure. Messages read by READUPDATE must be replied to by the REPLY procedure; messages read by READ are not replied to.

If <receive-depth> is more than 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

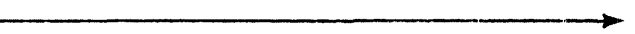
A process ID is used to open a file and to send messages to a designated process and, optionally, to wait for a reply.

A process ID has two forms:

If it references a process not in the PPD, it consists of:

```
<process-id>[0:2] = <creation-timestamp>
<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 the processor module where B is executing 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>
```

which is application-defined and entered into the PPD by the operating system at process creation time. A process name consists of a dollar sign ($) followed by up to five alphanumeric characters, the first of which must be alphabetic. A <cpu,pin> is allowed in this form of process ID so that a file can be opened using a process ID returned from NEWPROCESS, MOM, or LASTRECEIVE. The <cpu,pin> is ignored as far as determining the process to be opened. However, if it is included, it must contain a valid CPU number.

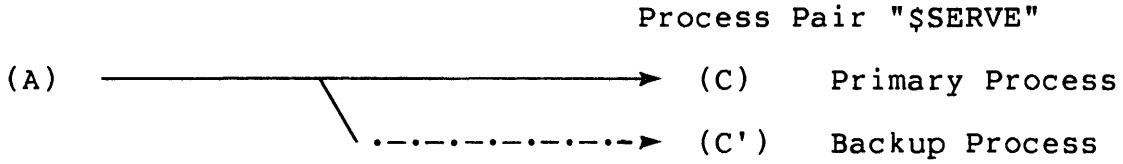
If the process name form of process ID is used to open a file, the file references the process pair. If the process opening the pair is not a member of the pair, the process name references the pair as a single entity. In this situation, the opener must specify, by using the <sync-depth> parameter to OPEN, whether or not the file system is to automatically redirect communication to the backup process in the event that the primary process fails.

If the <sync-depth> parameter is 1 or more, communication occurs with the primary process of the pair while it is operable. If it becomes inoperable, subsequent communication is redirected to the backup process (if any). This redirection of communication occurs in a manner invisible to processes outside of the pair. If the backup process does not exist and the primary process is inoperable, an error 201 is returned to the originator of a message.

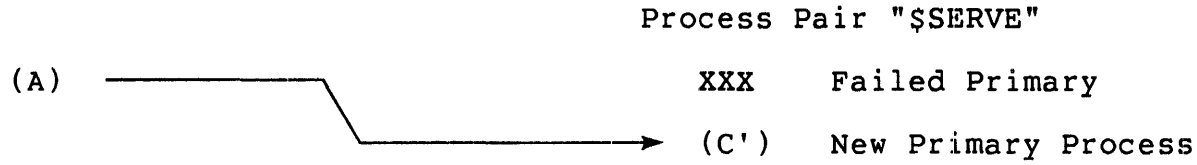
If the <sync-depth> parameter is 0, an error indication is returned at the first attempt to communicate with the process pair after a failure of the primary process. A subsequent retry by the application process causes the file system to redirect the retry and all further communication to the backup process.

For example, a process pair named "$SERVE" is opened with a sync depth of 1 (auto-retry by the file system):

```
INT .FNAME[0:11] := ["$SERVE", 9 *["  "]];

.
CALL OPEN ( FNAME, FNUM,, 1 );
```



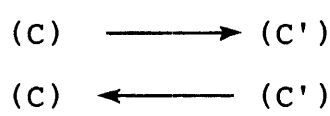
A opens the pair using the process name $SERVE. Communication occurs implicitly with the primary process C while it is operable. If C stops or if C's processor module fails, communication is redirected to C'. The failure of C and the redirection of communication to C' is invisible to A.



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 );
```



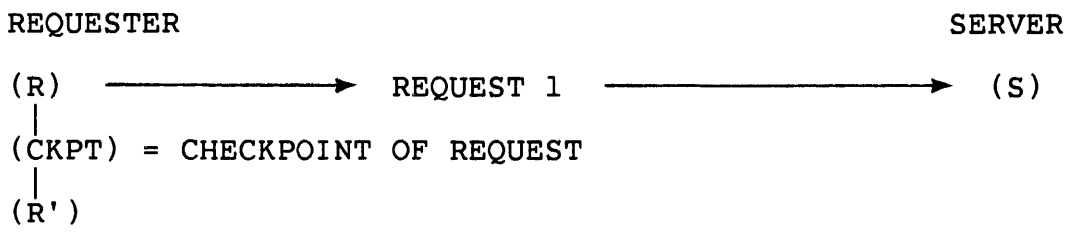
C opens the pair using the process name $SERVE. C communicates with its backup process C'. Likewise, C' opens the pair using the process name $SERVE. C' then communicates with its primary process C.

SYNC ID FOR DUPLICATE REQUEST DETECTION

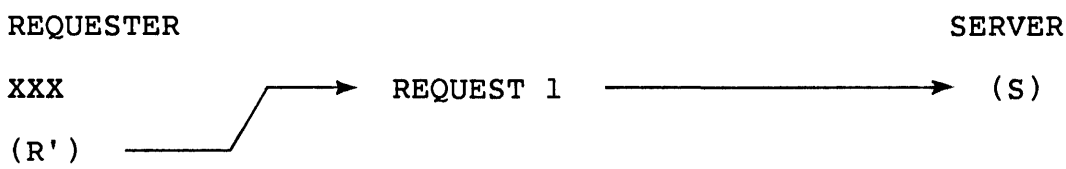
The sync ID scheme allows a server process (the process reading $RECEIVE) to detect duplicate requests from requester processes. Such duplicate requests are caused for two reasons:

1. By a backup requester process reexecuting the latest request of a failed primary requester process:

Normal:



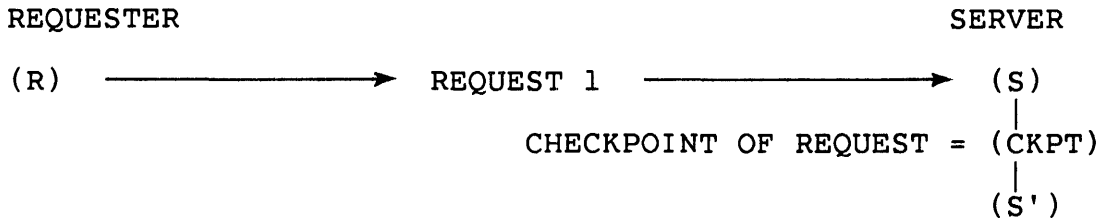
Failure of primary:



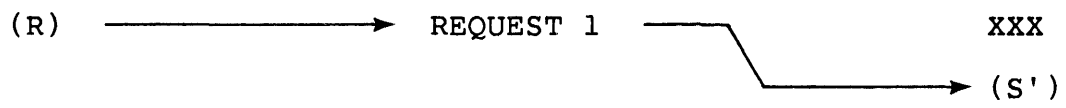
This results in the identical request being sent to the server. The server must recognize the request as a duplicate and return the latest reply for the requester.

2. By the file system reexecuting the latest request from a requester 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 requester.

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 access control block (ACB). Sync IDs 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 12). 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 sync ID value.

When a request is sent to a process (such as sending a CONTROL, CONTROLBUF, CLOSE, OPEN, SETMODE, WRITE, or WRITEREAD to a process file) the requester's sync ID is incremented by one just prior to the request being sent. (Therefore, a process's first sync ID subsequent to an open will have a value of zero.)

Note that neither a CANCEL or AWAITIO timeout completion has any effect on the sync ID. It will be an ever-increasing value.

Note also that the sync ID is independent of the <sync-depth> parameter to OPEN.

To understand the sync ID mechanism, you must consider the four possible processes involved in a request-reply transaction. You must also be familiar with checkpointing as described in Section 12. The following examples illustrate the four processes:

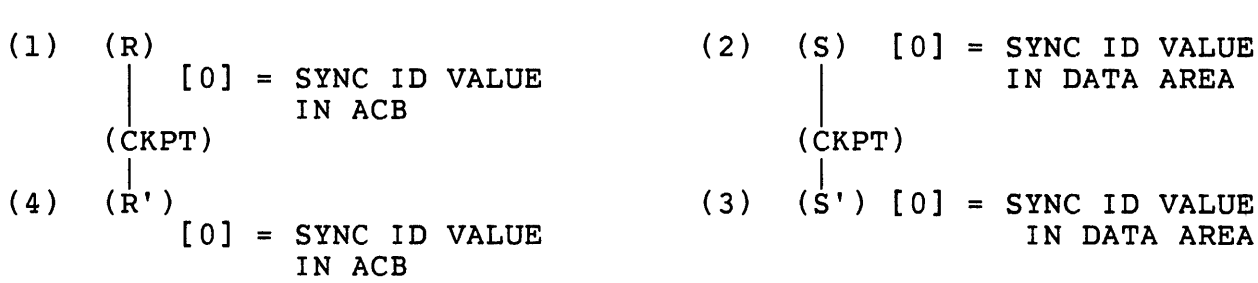
Following an open of the server by the requester, the sync ID values in all four processes are synchronized at 0:

1. The primary requester process R calls OPEN to establish communication with the server. This call to OPEN sets the sync ID value in the process file's ACB to 0.

2. The call to OPEN by the requester primary causes an OPEN system message to be sent to the primary server process S. The server calls RECEIVEINFO to obtain the sync ID value (which is 0). The server stores this value in a variable in its data area.

3. The primary server then checkpoints the sync ID and other information regarding the open to its backup S'; the backup server now has the sync ID value.

4. The primary requester process R calls CHECKOPEN to open a file in the backup requester process to the server. This call to OPEN sets the sync ID value in the process file's ACB in the backup server process to 0. (Note that this open also results in an OPEN system message being sent to the server.)



The following illustrates the completion of a successful transaction:

1. The transaction begins. The requester builds the request message, then checkpoints the request message and current sync ID value (now 0) to the backup requester.

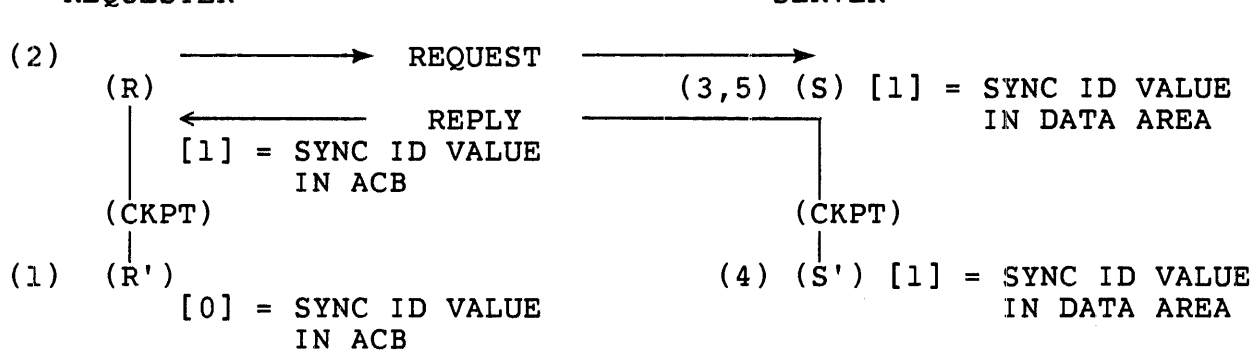
2. The requester sends the request message to the server. At this time the file system increments the sync ID value by 1.

3. The server picks up the request from $RECEIVE, then calls RECEIVEINFO to obtain the sync ID value (now 1).

4. The server examines the sync ID value for the requester to determine if it matches a request it has already received. Since 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 had matched, the sync ID saved by the server would have been returned to the requester.

5. The server then returns the reply value to the requester by a call to REPLY as shown below.



The following illustrates a transaction with failure of requester primary (sync ID = 1):

6. Transaction begins. The requester builds the request message, then checkpoints the request message and current sync ID value (now 1) to the backup requester.

7. The requester sends the request message to the server. At this time, the file system increments the sync ID value by 1. The sync ID value of the primary is now 2.

8. The server picks up the request from $RECEIVE, then calls RECEIVEINFO to obtain the sync ID value (now 2).

9. The server examines the sync ID value for the requester to determine if it matches a request it has already received. Since 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.

10. The server then returns the reply value to the requester by a call to REPLY.

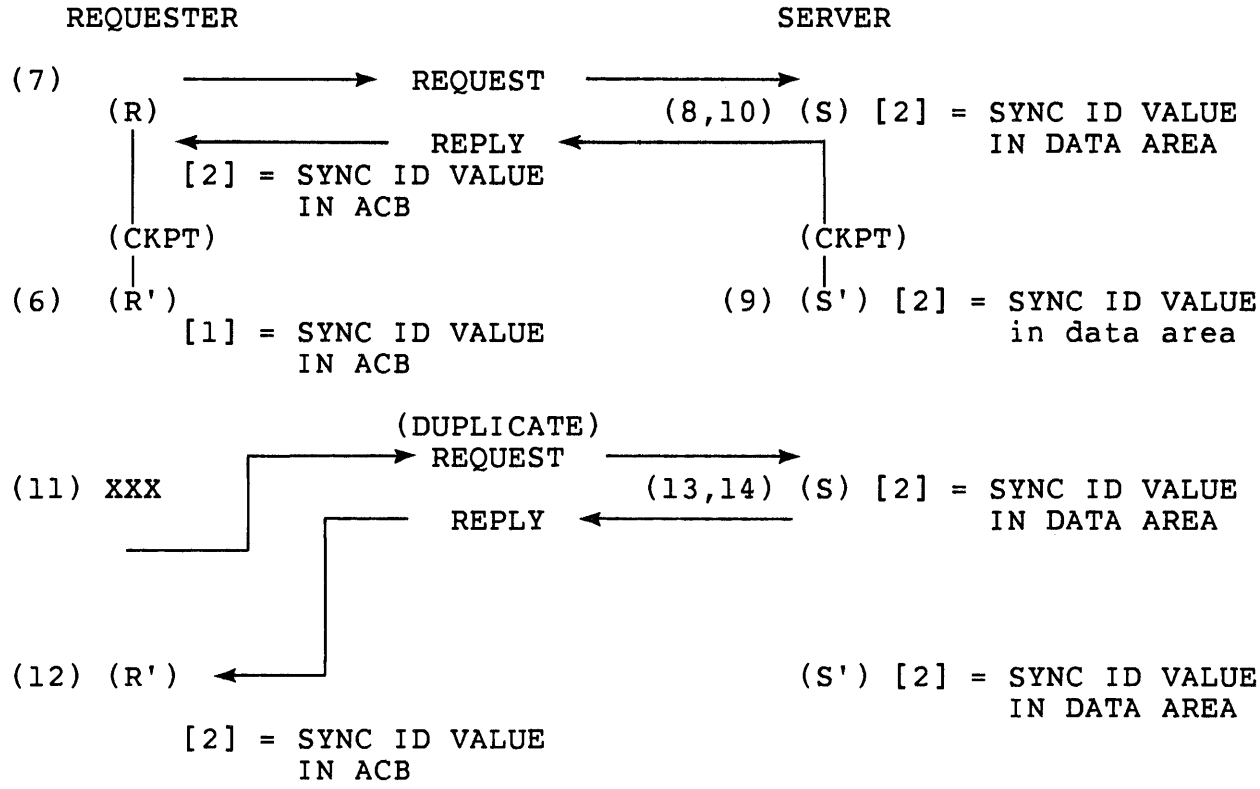
11. The primary requester fails. (Note that this example is valid no matter when the primary requester may fail).

12. The backup requester takes over and becomes the primary requester. It sends the latest request message that was checkpointed by the failed primary to the server. At this time the file system increments the sync ID value by 1. The sync ID value of the backup is now 2.

13. The server picks up the request from $RECEIVE, then calls RECEIVEINFO to obtain the sync ID value (now 2).

14. The server examines the sync ID value for the requester 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 requester, the server knows that it has already executed

this operation. Therefore, it returns the appropriate saved reply value for this request.



The following illustrates a transaction with failure of server primary (sync ID = 1):

6. Transaction begins. The requester builds the request message, then checkpoints the request message and current sync ID value (now 1) to the backup requester.

7. The requester 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 from $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).

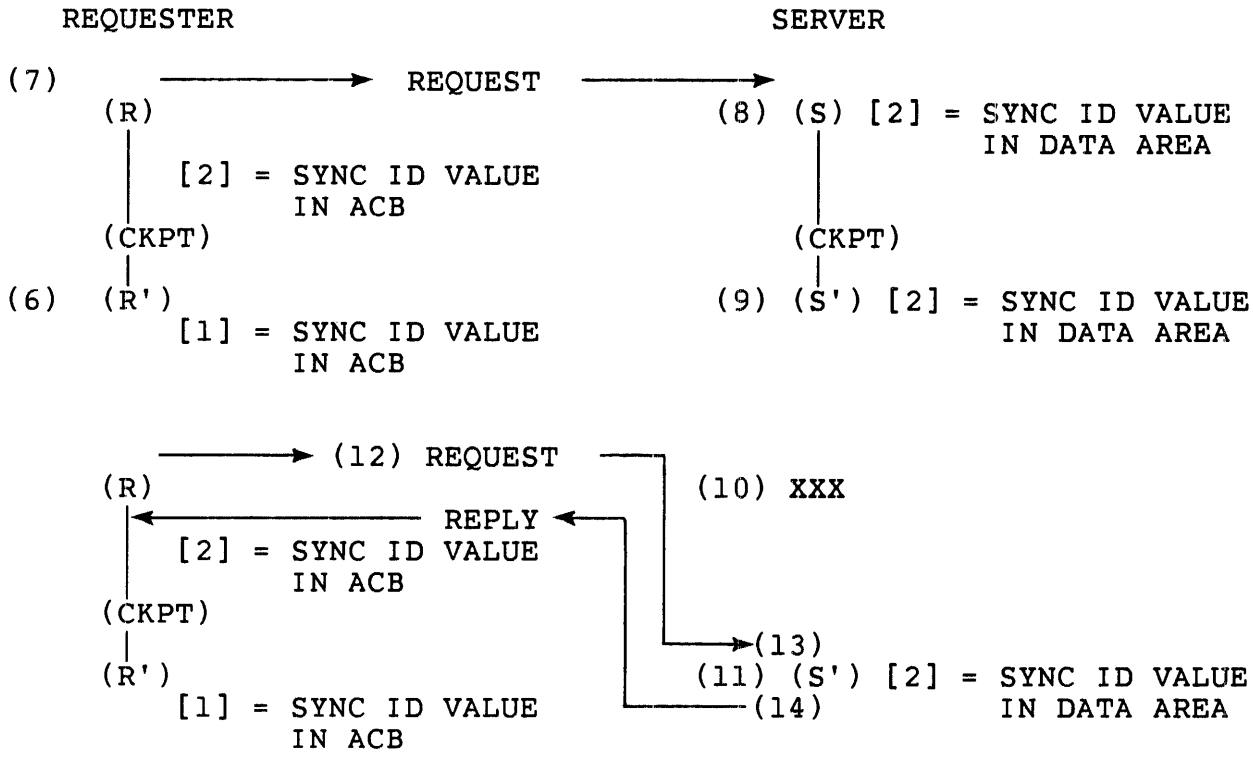
11. The backup server takes over and becomes the primary server. It executes the latest request that was checkpointed by the

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 and the failure is ignored.

12. The file system, on behalf of the requester process, retries the current request, sending it to the new primary server.

13. The server picks up the request from $RECEIVE, then calls RECEIVEINFO to obtain the sync ID value.

14. The server examines the sync ID value for the requester 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 requester, the server knows that it has already executed this operation. Therefore, it returns the appropriate saved reply value for this request.



Example code in requester and server:

```
   Requester                          Server

CALL OPEN(FN1,F1,...);
CALL CHECKOPEN(FN1,F1,...);
WHILE 1 DO                          WHILE 1 DO
 BEGIN                               BEGIN
    .                                 CALL READUPDATE(R,REQ);
    ! build request.                  CALL RECEIVEINFO(PID,,SID);
    .                                 IF SID = LAST^SID THEN
  CALL CHECKPOINT(STK,,F1,REQ,...);    ! duplicate request.
  CALL WRITEREAD(F1, REQ, ...);       CALL REPLY ( LAST^REPLY )
 END;                                ELSE
                                      BEGIN
                                        CALL CHECKPOINT(STK,REQ,);
                                        .
                                        ! process request.
                                        .
                                        LAST^REPLY := NEW^REPLY;
                                        LAST^SID := SID;
                                        CALL REPLY ( NEW^REPLY );
                                      END;
                                     END;
```

Additional information on checkpointing and fault-tolerant programming appears in Section 12.

INTERPROCESS COMMUNICATION EXAMPLE

The following is an example of a two-way transmission between a requester process and a server process. The server accepts OPEN and CLOSE system messages but treats CONTROL, SETMODE, and CONTROLBUF system messages as invalid operations. No message queueing is performed. Only one open is permitted for each requester process.

The following depicts the call in the requester 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 (sync depth = 1).
```

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, nowait 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 Section 12 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],           ! requester <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.
      BEGIN
        SYSTEM^MESSAGE := >; ! save system message condition.
        CALL RECEIVEINFO ( PID ,, SYNC^ID );
        IF SYSTEM^MESSAGE THEN
          CALL PROCESS^SYSTEM^MESSAGE (RECV^BUF,RECV^CNT,PID)
        ELSE
          CALL PROCESS^USER^REQUEST (RECV^BUF,RECV^CNT, PID ,
                                     SYNC^ID );
      END;     ! read a message.
  END;     ! loop on requests.
```

The following depicts the action in the server process when it receives system messages:

```
PROC PROCESS^SYSTEM^MESSAGE ( RECV^BUF, RECV^CNT, PID );
   INT .RECV^BUF,
        RECV^CNT,
       .PID;
```

```
  BEGIN
    INT REPLY^ERROR^CODE := 0,
        REQUESTER;

    CASE $ABS ( RECV^BUF ) - 30 OF
      BEGIN

! -30 ! ! OPEN system message.
         ! check for nowait I/O depth > 1.
       IF RECV^BUF[1].<12:15> > 1 THEN REPLY^ERROR^CODE := 28

            ! An attempt to open this process with the maximum
            ! number of concurrent operations > 1 is rejected with
            ! an error 28 indication.

            ELSE
            ! try to add opener to server's directory.
            IF NOT ADDPID ( PID ) THEN REPLY^ERROR^CODE := 12;

            ! The ADDPID procedure is used to add a new requester
            ! to the local directory.  If the requester's process
            ! ID is entered successfully (there is room in the
            ! directory), ADDPID returns a true value, and a
            ! successful open indication is returned to the
            ! opener.  Otherwise, ADDPID returns a false value,
            ! and a FILE IN USE error is returned to the opener
            ! (the open fails).

! -31 ! ! CLOSE system message.
            CALL DELPID ( PID );

            ! The DELPID procedure is used to delete an entry
            ! in the local directory.

! -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
              REQUESTER := LOOKUPPID ( PID );
```

The LOOKUPPID procedure is used to look up a requester process' sync ID in the local directory. If the PID exists, the entry number in the directory is returned. If not, a zero is returned. (If the requester is not found, you should supply error handling at this point.)

```
              SYNC^COUNT [REQUESTER] := 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 associated with the failing processor module from the local directory.

```
               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 requester 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;
       CALL WRITEREAD ( SFNUM, SEND^BUFFER, REQUEST^LEN,
                        REPLY^COUNT );
       IF < THEN ... ; ! fatal error.
     END;
```

And the corresponding receipt of the request in the server process:

```
   ! global variables
   LITERAL
        MAX^REQSTRS =  16, ! maximum number of requesters
                           ! allowed.
        REPLY^SIZE  = 256; ! reply message size.

   INT(32)
        ! sync count for duplicate request detection.
       .SYNC^COUNT [1 : MAX^REQSTRS] := MAX^REQSTRS * [0D];
```

```
     ! reply for each requester.
INT .REPLY^BUF  [REPLY^SIZE :
                 MAX^REQSTRS * REPLY^SIZE + REPLY^SIZE - 1],
     ! reply error code for each requester.
    .REPLY^ERROR^CODE [1 : MAX^REQSTRS],
     ! reply length for each requester.
    .REPLY^LEN [1 : MAX^REQSTRS];

PROC PROCESS^USER^REQUEST (RECV^BUF,RECV^CNT,PID , SYNC^ID );
   INT .RECV^BUF,
       RECV^CNT,
      .PID,     ! PID of requester.
  INT(32) SYNC^ID; ! sync ID of request.
  BEGIN
    INT REQUESTER;

    REQUESTER := LOOKUPPID ( PID );
```

The LOOKUPPID procedure is used to look up a requester process sync in a local directory. If the PID exists, the entry number in the directory is returned. If not, a zero is returned. (If the requester is not found, you should supply error handling at this point.)

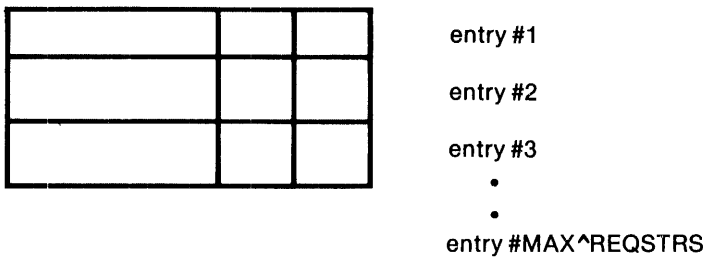
```
    ! check for duplicate request.
    IF SYNC^ID <> SYNC^COUNT [REQUESTER] THEN ! new request
      BEGIN
        ! save sync count of current requester.
        SYNC^COUNT [REQUESTER] := SYNC^ID;

         .
         .   The request is processed.
         .

        ! save the reply.
        REPLY^BUF [REQUESTER * REPLY^SIZE] ':='
           RESULT FOR RESULT^LEN;
        REPLY^LEN [REQUESTER]         :=  RESULT^LEN;
        REPLY^ERROR^CODE [REQUESTER] :=  RESULT^ERROR;
      END;

    ! return the reply to the requester.
    CALL REPLY ( REPLY^BUF [REQUESTER * REPLY^SIZE],
                 REPLY^LEN [REQUESTER],
                 ,
                 ,
                 REPLY^ERROR^CODE [REQUESTER] );
```

If this is a duplicate request, the last reply is returned to the requester.

```
  END; ! PROCESS^USER^REQUEST.
```

The following are the procedures in the server process that maintain the local directory of process IDs. The directory is of the form

```
INT .PIDS[5:MAX^REQSTRS*5 + 5] := (MAX^REQSTRS * 5) * [0];
```




```
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) <= MAX^REQSTRS DO
      IF PID = PIDS[ENTRYNO * 5] FOR COMP^LEN THEN ! found it.
         RETURN ENTRYNO;

    RETURN 0; ! not found.
  END; ! lookuppid.
```

```
INT PROC ADDPID(PID);
   INT .PID;

  ! return values.
  !  0 = PID not added to directory.
  ! >0 = entry no. of PID in directory.

  BEGIN
    INT ENTRYNO,                    !entry in local PID directory.
        ZERO[0:3] := [0,0,0,0]; ! for lookup of empty
                                ! directory slot

    IF (ENTRYNO := LOOKUPPID(PID)) THEN !already in directory.
      BEGIN
        ! check for duplicate open.
        IF PIDS[ENTRYNO * 5 + 3] <> PID[3] AND
           PIDS[ENTRYNO * 5 + 4] <> PID[3] THEN  !first open
           PIDS[ENTRYNO * 5 + 4] := PID[3];      !by backup.
      END
    ELSE ! not in directory.  First open by PID
      BEGIN
        IF (ENTRYNO := LOOKUPPID(ZERO)) THEN ! look for empty
                                             ! slot.
          BEGIN
            PIDS[ENTRYNO * 5] ':=' PID FOR 4;
            SYNC^COUNT[ENTRYNO] := -1D; ! init req. sync^count
          END;
      END;
    RETURN ENTRYNO; ! returns zero if no room in directory.
  END; ! ADDPID.

PROC DELPID(PID);
  INT .PID; ! PID to be deleted.

  BEGIN
    INT ENTRYNO; ! entry number in local PID directory.

    IF (ENTRYNO := LOOKUPPID(PID)) THEN  ! delete it.
      IF PIDS[ENTRYNO * 5 + 4] THEN !was open by process pair.
        BEGIN
          IF PIDS[ENTRYNO * 5 + 3] = PID[3] THEN   ! close by
                                                   ! primary
            ! replace primary entry with backup.
            PIDS[ENTRYNO * 5 + 3] := PIDS[ENTRYNO * 5 + 4];
          ! clear backup entry.
          PIDS[ENTRYNO * 5 + 4] := 0;
        END
      ELSE ! was open by one process.
        PIDS[ENTRYNO * 5] ':=' [0,0,0,0];
  END; ! DELPID.
```

```
PROC DELALLPIDS(CPU);
   INT CPU; ! processor module number of PIDs to be deleted.

  BEGIN
    INT ENTRYNO := 0, ! entry in local PID directory.
        temp;

    WHILE (ENTRYNO := ENTRYNO + 1) <= MAX^REQSTRS 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.
```

ERROR RECOVERY

For the $RECEIVE file, there are no error conditions for which error recovery should be attempted, except error 40 (timeout).

For a process file opened with a sync depth greater than zero, there are no error conditions for which error recovery should be retried, except error 40.

For a process file opened with a sync depth of zero, an operation that 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 is returned on the reexecution of the call. At this point, the error can be considered fatal.

SECTION 5

INTERFACING TO THE GUARDIAN COMMAND INTERPRETER

This section describes the interface to the GUARDIAN operating system through the GUARDIAN command interpreter (COMINT).

GENERAL CHARACTERISTICS OF THE COMMAND INTERPRETER

The GUARDIAN command interpreter provides a direct interface between system users and the GUARDIAN operating system. Users at on-line terminals interact with the command interpreter by typing in commands. If a command is given to run a program, then the program begins executing. Such a program may be either a user-written program or a utility program supplied by Tandem, such as FUP, PUP, BACKUP, RESTORE, SPOOLCOM, and so on.

If a command is given to perform some specific operation, the necessary system functions are executed. After completing the command, the command interpreter then asks the user for another command (by displaying a colon ":" on the terminal).

Some functions that the command interpreter performs are:

- List disc file names
- Create, rename, and purge disc files
- Set default disc volume/subvolume names and default security
- Run and pass parameters to processes
- Put a process into the debug state
- Stop process execution

Most of these functions do not directly affect the design of your application program. You must, however, be aware of how the command interpreter passes parameters to your application processes and how the default volume and subvolume names are used. (For a description of the command interpreter functions, see the GUARDIAN Operating System User's Guide).

An additional consideration is that the command interpreter makes use of the BREAK feature on the home terminal. Because of this, any application process that is run with the command interpreter and also uses BREAK on the home terminal, must do so in a proper manner. See "Using BREAK (Multiple Processes per Terminal)" in Section 6.

PASSING RUN-TIME PARAMETER INFORMATION TO AN APPLICATION PROCESS

Application-dependent parameter information can be specified prior to and at the same time as the command is given to run a program. This information is sent to the new process in the form of one or more interprocess messages.

There are six command interpreter commands that can affect the parameter information to be sent:

1. The VOLUME command specifies the default volume and subvolume names to be passed to the new process. Network volume names cannot exceed six characters, excluding the dollar sign "$".

2. The RUN command specifies the IN and OUT files and optional parameter string to be passed to the new process.

   The default volume and subvolume names, the IN and OUT file names, and the optional parameter string are passed to your application process in the startup message. If the network form of the object file is given, the network form of the volume IN and OUT file names indicating the node number of the current default node, will be in the startup message.

3. The ASSIGN command is used to make logical file assignments for programs written in such languages as COBOL or FORTRAN. A logical file assignment equates a Tandem file name with a logical file of a program and, optionally, assigns file characteristics to that file. For each ASSIGN in effect when a program is run, one assign message containing the assignment parameters is sent at the option of the new process. This follows the transmission of the startup message.

4. The PARAM command is used to associate an ASCII value with a parameter name. This command is typically used by languages

such as COBOL or FORTRAN to give initial values to program variables. If any PARAMs are in effect when a program is run, a single param message containing the parameter names and values is sent at the option of the new process. This follows the transmission of any assign messages (if requested).

5. The CLEAR command is used to clear ASSIGN and PARAM settings.

6. The SYSTEM command, used with a nonblank system name, implicitly causes remote programs to be run and the network form of the volume IN and OUT file names to be passed in the startup message.

NOTE

If your process opens the $RECEIVE file and specifies that it wishes to receive OPEN, CONTROL, SETMODE, and CLOSE system messages, the first message it receives will be an OPEN message. This is followed by the startup message and then by any assign messages and/or a param message. The final message is a CLOSE message (the OPEN and CLOSE messages are caused by the command interpreter opening and closing the new process).

The VOLUME, RUN, ASSIGN, PARAM, CLEAR, and SYSTEM commands are documented in the GUARDIAN Operating System Utilities Reference Manual.

The startup, assign, and param messages associated with the above commands are of particular importance to users of both FORTRAN and COBOL. The routines in the Saved Message Utility can be used to check for the existence of initial process creation messages and to retrieve, replace, or delete portions of them. Then by using the CREATEPROCESS routine, you can create a new process with its own set of assign, param, and startup messages. Refer to the Saved Message Utility in the FORTRAN or COBOL reference manual.

Startup Message

The open and startup messages are sent to the new process immediately following the successful creation of the new process. The open and startup messages are read by the process from its $RECEIVE file.

The form of the startup parameter message is:

```
STRUCT CI^STARTUP;
  BEGIN                          !  word
    INT MSGCODE;                 !  [0] -1.
    STRUCT DEFAULT;
      BEGIN
        INT VOLUME [0:3],        !  [1] $<default-volume-name>.
            SUBVOL [0:3];        !      <default-subvol-name>.
      END;
    STRUCT INFILE;
      BEGIN
        INT VOLUME [0:3],        !  [9] IN parameter <filename>
            SUBVOL [0:3],        !      of RUN command.
            DNAME  [0:3];
      END;
    STRUCT OUTFILE;
      BEGIN
        INT VOLUME [0:3],        ! [21] OUT parameter <filename>
            SUBVOL [0:3],        !      of RUN command.
            DNAME  [0:3];
      END;
    STRING PARAM [0:n-1];        ! [33] <parameter-string> (if any)
  END; ! CI^STARTUP^MSG.         !      of RUN command.  This is in
                                 !      either of the following
                                 !      forms:
                                 !
                                 !          <parameter-string>
                                 !            <null>[<null>]
                                 !     or
                                 !          <null><null>
                                 !
                                 !          <n> = ( <count read> - 66 )
```

The maximum length possible for a startup message is 596 bytes (including the trailing null characters). The parameter message length is always an even number. If necessary, the command interpreter pads the <parameter-string> with an additional null.

The following is an example showing an application process reading its startup message.

First, the following VOLUME command is entered:

```
:VOLUME $STORE1.ACCTRCV
```

Then the following RUN command is given:

```
:RUN XNSTP/IN INFILE, OUT OUTFILE, NAME/ 1,10,BYTE,DESCENDING
```

The parameter NAME without a corresponding <process-name> causes the system to create a name for the new process. The name is entered into the PPD if the process is created successfully.

The command interpreter first attempts to run the program indicated by the expanded form of XNSTP--$STORE1.ACCTRCV.XNSTP. If the new process is created, the command interpreter forms a message to be sent to the new process from the current default volume and subvolume names, the IN parameter information, the OUT parameter information, and the application-dependent parameter string. The message contains the following information:

```
word[0]  = - 1                          ! means start-up message.
word[1]  = "$STORE1 "                   ! $<default-volume-name>.
word[5]  = "ACCTRCV "                   ! <default-subvol-name>.
word[9]  = "$STORE1 ACCTRCV INFILE  " ! IN param <filename>.
word[21] = "$STORE1 ACCTRCV OUTFILE " ! OUT param <filename>.
word[33] = "1,10,BYTE,DESCENDING"     ! <parameter-string>.
word[43] = <null><null>                 ! null terminators.
```

One of the first actions the XNSTP program must perform is to open and read the $RECEIVE file:

```
INT .RECEIVE[0:11] := ["$RECEIVE", 8 * ["  "]], ! data
     RECV^FNUM,                                 ! declarations
     IN^FNUM,
     OUT^FNUM,
     NUM,
    .BUFFER[0:99],
     NUM^READ,
    .CREATOR[0:3],
    .LASTPID[0:3];
LITERAL
    RCV^FLAGS  = %40000,
    RCV^DEPTH  = 1;
STRING .PARMS[0:39], .SBUFFER := @BUFFER '<<' 1;
   .
CALL OPEN(RECEIVE, RECV^FNUM, RCV^FLAGS, RCV^DEPTH);
   .
CALL READUPDATE(RECV^FNUM, BUFFER, 200, NUM^READ);
   .
```

The application program assumes that the first message is the open message:

```
CALL REPLY ( ,,,, 0);
```

The application program then reads the startup message:

```
CALL READUPDATE(RECV^FNUM, BUFFER, 200, NUM^READ);
```

The application program then ensures that the incoming message is the startup message:

```
   .
IF BUFFER <> -1 THEN CALL ABEND;
   .
```

The application process opens its input and output files using the information passed in the parameter message:

```
   .
CALL OPEN(BUFFER[9],in^fnum);
   .

opens "$STORE1 ACCTRCV INFILE"

   .
CALL OPEN(BUFFER[21], OUT^FNUM);
   .

opens "$STORE1 ACCTRCV OUTFILE"
```

then saves the <parameter-string> information:

```
   .
NUM := NUM^READ - 66; ! length of parameter string in bytes.
IF NUM <= 40 THEN     ! parameter string will fit, move it in.
   PARMS ':=' SBUFFER[66] FOR NUM
ELSE                  ! parameter string too long.
   .
```

Assign Message

One assign message is optionally sent to the new process for each assignment in effect at the time of the creation of the new process. Assign messages are sent immediately following the startup message if the process does either one of the following:

- The process replies to the startup message with an error return value of REPLY = 70 (CONTINUE). The command interpreter then sends both assign and param messages.
- The process replies to the startup message with an error return value of 0, but with a reply of one to four bytes, and bit 0 of the first byte of the reply is set to 1. The command interpreter also sends param messages if bit 1 of the first byte of the reply is set to 1.

The form of the assign message is:

```
STRUCT CI^ASSIGN;                 ! assign message.
  BEGIN                           !
    INT MSG^CODE;                 !  [0] -2
                                  !
    STRUCT LOGICALUNIT;           ! PARAMETERS TO ASSIGN COMMAND.
      BEGIN                       !
        STRING PROGNAMELEN,       !  [1] name length, 0:31 bytes
               PROGNAME[0:30],    !  <program-unit> | * } <blanks>
               FILENAMELEN,       ! [17] name length, 0:31 bytes
               FILENAME[0:30];    !      <logical-file><blanks>
      END;                        !

    INT(32) FIELDMASK;            ! [33] bit mask to indicate
                                  ! which of the following fields
                                  ! were supplied (1 = supplied):
                                  !      .<0> = <Tandem-filename>
                                  !      .<1> = <pri-extent-size>
                                  !      .<2> = <sec-extent-size>
                                  !      .<3> = <file-code>
                                  !      .<4> = <exclusion-size>
                                  !      .<5> = <access-spec>
                                  !      .<6> = <record-size>
                                  !      .<7> = <block-size>
                                  !
    STRUCT TANDEMFILENAME;        ! [35] <Tandem-filename>
      BEGIN                       !
        INT VOLUME  [0:3],        !
            SUBVOL  [0:3],        !
            DFILE   [0:3];        !
      END;                        !
    ! CREATESPEC                  !
    INT PRIMARYEXTENT,            ! [47] <pri-extent-size>.
        SECONDARYEXTENT,          ! [48] <sec-extent-size>.
        FILECODE,                 ! [49] <file-code>.
        EXCLUSIONSPEC,            ! [50]   %00 if SHARED,     }
                                  !        %20 if EXCLUSIVE,  }
                                  !        %60 if PROTECTED.  }
        ACCESSSPEC,               ! [51] %0000 if I-O,        }
                                  !      %2000 if INPUT,      }
                                  !      %4000 if OUTPUT.     }
                                  !   [50-51] corresponds to  }
                                  !      flag param of OPEN.  }
        RECORDSIZE,               ! [52] <record-size>.
        BLOCKSIZE;                ! [53] <block-size>.
  END;
```

The length of this message is 108 bytes.

Param Message

A param message is optionally sent to the new process if any parameters are in effect at the time of the creation of the new process. The param message is sent immediately following any assign messages if the process does either one of the following:

- The process replies to the startup message with an error return value of REPLY = 70 (CONTINUE). The command interpreter then sends both assign and param messages.
- The process replies to the startup message with an error return value of 0, but with a reply of one to four bytes, and bit 1 of the first byte of the reply is set to 1. The command interpreter also sends assign messages if bit 0 of the first byte of the reply is set to 1.

The form of the param message is:

```
STRUCT CI^PARAM;                    ! param message.
  BEGIN                             !
    INT MSG^CODE,                   !  [0] -3
        NUMPARAMS;                  !  [1] number of parameters
                                    !  included in this message.
    STRING PARAMETERS [0:1023];     !  [2] beginning parameters.
  END;
```

The field PARAMETERS in the above message format is comprised of NUMPARAMS records of the form (offsets are given in bytes):

```
<PARAM>[0]          = "n", length in bytes of <parameter-name>
<PARAM>[1]   FOR n  = <parameter-name>
<PARAM>[n+1]        = "v", length in bytes of <parameter-value>
<PARAM>[n+2] FOR v  = <parameter-value>
```

The maximum length of this message is 1028 bytes.

Reading All Parameter Messages

If you want to read all of the parameter messages, you must consider the following:

1. To indicate to the command interpreter that all current parameter information is desired, the application process must reply to the startup message. Therefore, the startup message must be read by a call to READUPDATE so that a subsequent reply can be made. This means that the $RECEIVE file must be opened with:

   OPEN <receive-depth> >= 1

2. The command interpreter indicates the end of the series of parameter messages by closing its file to the application process. Therefore, the application process must open the $RECEIVE file with:

   OPEN <flags>.<1> = 1

   so that it will receive OPEN and CLOSE system messages.

Your application process receives the following sequence of messages when reading all parameter messages:

1. OPEN system message (message code = -30)

2. Startup message (message code = -1)

Your process must reply with REPLY <error-return> = 70, (or with a buffer at least one byte long, with bits <0> and <1> both set to 1).

3. Zero or more assign messages (message code = -2)

4. Zero or one param message (message code = -3)

5. CLOSE system message (message code = -31)

The general sequence to read the parameter messages is shown in the following example:

```
PROC READ^PARAMETER^MESSAGES;
  BEGIN
    INT .RCV^FNAME [0:11] := ["$RECEIVE", 8 * ["  "]],
        RCV^FNUM,
        .RCV^BUF   [0:514]
        CNT^READ,
        REPLY^CODE        := 0;

    LITERAL
        RCV^FLAGS  = %40000, ! OPEN-CLOSE messages.
        RCV^DEPTH  =      1, ! READUPDATE-REPLY.
        RCV^CNT    =   1030, !
        CLOSE^MSG  =    -31; ! CLOSE message code.

    ! open $RECEIVE.
    CALL OPEN ( RCV^FNAME, FCV^FNUM, RCV^FLAGS, RCV^DEPTH );
    IF <> THEN ...;
    ! read open message.
    CALL READUPDATE ( RCV^FNUM, RCV^BUF, RCV^CNT, CNT^READ );
    WHILE RCV^BUF <> CLOSE^MSG DO
      BEGIN
        CASE $ABS ( RCV^BUF ) OF
          BEGIN
    !    0 ! ;
    !   -1 ! BEGIN ! startup message.
                .
              REPLY^CODE  := 70;
                  process startup message.
                .
              END;
    !   -2 ! BEGIN ! assign message.
                .
              REPLY^CODE  := 0;
                  process assign message.
                .
              END;
    !   -3 ! BEGIN ! param message.
                .
              REPLY^CODE  := 0;
                  process param message.
                .
              END;
              OTHERWISE;
            END;
        CALL REPLY ( ,,,, REPLY^CODE );
        CALL READUPDATE ( RCV^FNUM,RCV^BUF,RCV^CNT,CNT^READ );
      END; ! while not CLOSE^MSG.
    ! close $RECEIVE.
    CALL REPLY (,,,,0); ! reply to close^msg.
    CALL CLOSE ( RCV^FNUM );
  END; ! read^parameter^messages.
```

APPLICATION PROCESS TO COMMAND INTERPRETER INTERPROCESS MESSAGES

There are two messages that the command interpreter accepts from application processes:

- The wakeup message (message code = -20)
- The display message (message code = -21)

An interprocess message is sent to a particular command interpreter by opening the CRTPID of that command interpreter and then writing the message using the WRITE procedure (see "Processes (Interprocess Communication)" in Section 2).

Wakeup Message

The wakeup message, when received by a command interpreter, causes that command interpreter, if it is currently in the paused state, to return from the paused state to the command input mode (to "wake up").

If the command interpreter is not in the paused state (if it is prompting for a command or executing a command other than RUN), a wakeup message is ignored.

The form of the wakeup message is:

```
STRUCT WAKEUP^MSG;
  BEGIN
    INT MSGCODE; ! -20
  END;
```

The length of this message is two bytes.

The intended use of this message is to allow a process that is a descendant of a command interpreter to wake up that command interpreter. A typical case is with a nonnamed process pair: the backup process calls STEPMOM with the primary process the object of the call (so that the backup will know if the primary fails); the call to STEPMOM cancels the primary process's relationship with the command interpreter. The primary process, just prior to stopping, sends a wakeup message to the command interpreter, since the stop message will be sent to the backup.

Display Message

The display message, when received by a command interpreter, causes the command interpreter to display the text contained in the message. The text is displayed just prior to the next time the command interpreter prompts for a command with a ":".

A command interpreter has the capability of storing up to eight 132-byte display messages until it is able to display the message text. If this buffer is full when another display message is sent to it, the incoming display message is rejected with an error 45 (FILE IS FULL).

The form of the display message is:

```
STRUCT DISPLAY^MSG;
  BEGIN
    INT MSGCODE;            ! -21
    STRING TEXT [0:n-1]; ! n <= 132.
  END;
```

The length of this message is (2 + display text length) bytes. The length of the text portion is implied in the write count used to send this message.

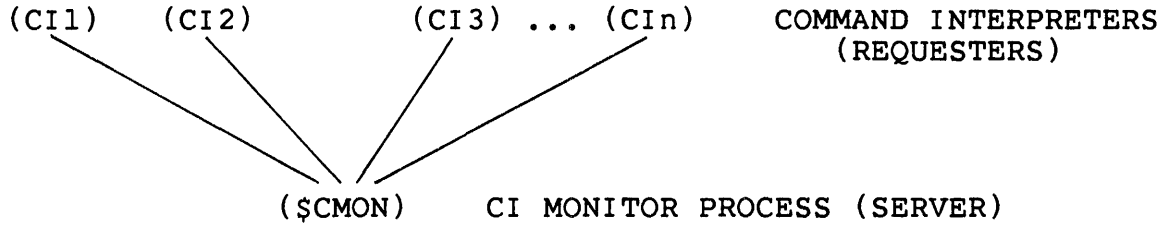
USER-SUPPLIED CI MONITOR PROCESS ($CMON)

The purpose of a user-supplied command interpreter monitor process is to allow users to control and monitor:

- Logons and logoffs
- Adding or deleting users
- Changing user passwords
- Altering run-time priorities
- Illegal logon attempts

To provide these capabilities, the command interpreter opens a process named $CMON. The command interpreter notifies the $CMON process (by interprocess message) each time one of the following commmands is given: ADDUSER, ALTPRI, DELUSER, LOGOFF, LOGON, PASSWORD, REMOTEPASSWORD, or implicit or explicit RUN commands.

The relationship between all command interpreters in the system and the $CMON process is that of requesters and server, respectively. (See Section 4.)



The $CMON process reads the notification messages from its $RECEIVE file. The $CMON process must then reply to each message by rejecting the command (in which case the command is not executed); accepting the command (the command will be executed as is); or in the case of the RUN command, modifying the command by specifying a different program file, a different processor module for execution, or a different execution priority.

This control is implemented by several interprocess messages to the $CMON process and several possible replies that the $CMON process may make to the requesting command interpreter in response. The $CMON process must call the REPLY procedure to make its responses. Therefore, its $RECEIVE file must be open with a receive depth >= 1, and the notification messages must be read by making calls to READUPDATE.

Communication Between Command Interpreters and $CMON

The interprocess messages from a command interpreter to the $CMON process include:

- Logon message
- Logoff message
- Process creation message (such as the RUN command)
- Illegal logon attempts message
- Add user message
- Delete user message
- Alter run-time priorities message
- Change user passwords message

All messages are sent to $CMON by command interpreters on a nowait basis. If a message cannot be sent or if $CMON does not reply, the command interpreter closes the open sent to $CMON and proceeds to execute the command as if the $CMON process did not exist.

If the BREAK key is pressed while a message is outstanding to $CMON and the owner of BREAK is not a super-ID user, the message is cancelled and the command is aborted. If the BREAK key is pressed while a message is outstanding, and the owner of BREAK is the super ID user, the message is cancelled and the command is executed.

If the command interpreter encounters an I/O error when communicating with $CMON, it closes its open to $CMON and no longer attempts communication. The command interpreter tries to reopen $CMON (and attempts communication) when the next monitored command is issued.

$CMON Messages

The logon message is sent every time COMINT attempts to log on. The $CMON reply indicates whether the user is allowed to log on, and contains an optional display message. If the $CMON process is not running, then no $CMON logon restrictions will be in effect; all valid users may log on.

The form of the logon message is:

```
STRUCT  logon^msg;
  BEGIN
  INT    msgcode;              ! [0] -50
  INT    userid;               ! [1] user ID of user logging on.
  INT    cipri;                ! [2] initial priority of CI.
  INT    ciinfile[0:11];       ! [3] name of CI's command file.
  INT    cioutfile[0:11];      ! [15] name of CI's list file.
  END;
```

The form of the reply to the logon message is:

```
STRUCT  logon^reply;
  BEGIN
  INT     replycode;           ! [0] 0=allow LOGON,
                               !     1=disallow LOGON.
  STRING  replytext[0:n];      ! [1] optional message to be
                               !     printed.  Maximum of 132
                               !     bytes.
  END;
```

The logoff message is sent every time a user logs off. A separate logoff message is also sent when a user logs on without first logging off (implicit logoff). The $CMON reply contains an optional display message. If the $CMON process is not running, then COMINT does not attempt to write the logoff message.

The form of the logoff message is:

```
STRUCT  logoff^msg;
  BEGIN
  INT   msgcode;               ! [0] -51
  INT   userid;                ! [1] user ID of user logging off.
  INT   cipri;                 ! [2] initial priority of CI.
  INT   ciinfile[0:11];        ! [3] name of CI's command file.
  INT   cioutfile[0:11];       ! [15] name of CI's list file.
  END;
```

The form of the reply to the logoff message is:

```
STRUCT  logoff^reply;
  BEGIN
  INT     replycode;           ! [0] ignored by CI.
  STRING  replytext[0:n];      ! [1] optional message to be
                               !     printed.  Maximum of 132
                               !     bytes.
  END;
```

The process creation message is sent to $CMON whenever the user attempts to start a process, either explicitly (:RUN <prog-file>) or implicitly (:<prog-file>). Two forms of the $CMON reply message are recognized. The first is approval to run the process, and contains the <priority> and <cpu> in which to run. The second message aborts the new process creation attempt and displays an optional error message. If the $CMON process is not running, the user process will be started in the same CPU as COMINT's primary process and with a priority of 1 less than the current COMINT.

The run parameters IN <file>, OUT <file>, LIB <file> and SWAP <file> are included in the process creation message sent to $CMON.

The form of the process creation message is:

```
STRUCT  processcreation^msg;
  BEGIN
  INT    msgcode;              ! [0] -52
  INT    userid;               ! [1] user ID of user logged on.
  INT    cipri;                ! [2] initial priority of CI.
  INT    ciinfile[0:11];       ! [3] name of CI's command file.
  INT    cioutfile[0:11];      ! [15] name of CI's list file.
  INT    progname[0:11];       ! [27] expanded program file
                               !      name.
  INT    priority;             ! [39] the value of the PRI run
                               !      parameter if supplied;
                               !      otherwise -1.
  INT    processor;            ! [40] the value of the CPU run
                               !      parameter if supplied;
                               !      otherwise -1.
  INT    proginfile[0:11];     ! [41] the expanded IN file run
                               !      parameter if supplied;
                               !      otherwise the default IN
                               !      file.
  INT    progoutfile[0:11];    ! [53] the expanded OUT file run
                               !      parameter if supplied;
                               !      otherwise the default OUT
                               !      file.
  INT    proglibfile[0:11];    ! [65] the expanded LIB file run
                               !      parameter if supplied;
                               !      otherwise blanks.
  INT    progswapfile[0:11];   ! [77] the expanded SWAP file run
                               !      parameter if supplied;
                               !      otherwise blanks.
  END;
```

The forms of the reply to the process creation message are:

```
STRUCT  processcreation^reply;
  BEGIN
  INT     replycode;          ! [0] 0=create the process.
  INT     progname[0:11];     ! [1] expanded name of program
                              !     file to be run.
  INT     priority;           ! [13] execution priority of new
                              !      process or -1.  If -1,
                              !      then one less than the
                              !      current CI's priority is
                              !      used.
  INT     processor;          ! [14] processor where new
                              !      process is to run or -1.
                              !      If -1, then the current
                              !      CI's primary processor is
                              !      used.
  END;
```

or

```
STRUCT  processcreation^reply;
  BEGIN
  INT     replycode;          ! [0] 1=disallow process
                              !       creation.
  STRING  replytext[0:n];     ! [1] optional message to be
                              !     printed.  Maximum of 132
                              !     bytes.
  END;
```

The illegal logon message is sent on each failed attempt following two consecutive failed LOGON attempts. The $CMON reply message contains an optional display message. If the $CMON process is not running, no illegal logon message is sent.

The form of the illegal logon message is:

```
STRUCT  illegal^logon^msg;
  BEGIN
  INT     msgcode;            ! [0] -53
  INT     userid;             ! [1] user ID of user trying to
                              !     LOGON.
  INT     cipri;              ! [2] initial priority of CI.
  INT     ciinfile[0:11];     ! [3] name of CI's command file.
  INT     cioutfile[0:11];    ! [15] name of CI's list file.
  STRING  logonstring[0:n];   ! [27] the attempted LOGON
                              !      command string.  Maximum
                              !      of 132 bytes.
  END;
```

The form of the reply to the illegal logon message is:

```
STRUCT  illegal^logon^reply;
  BEGIN
  INT     replycode;          ! [0] ignored by CI.
  STRING  replytext[0:n];     ! [1] optional message to be
                              !     printed.  Maximum of 132
                              !     bytes.
  END;
```

The add user message is sent whenever the user attempts to add a user to the system. The $CMON reply indicates whether the user can be added, and contains an optional display message. If $CMON is not running, then no $CMON add user restrictions will be in effect. Any user can be added to the group if the current user is the group manager. Note that a super ID can add any user.

The form of the add user message is:

```
STRUCT  adduser^msg;
  BEGIN
  INT     msgcode;            ! [0] -54
  INT     userid;             ! [1] user ID of user adding the
                              !     user.
  INT     cipri;              ! [2] initial priority of CI.
  INT     ciinfile[0:11];     ! [3] name of CI's command file.
  INT     cioutfile[0:11];    ! [15] name of CI's list file.
  INT     groupname[0:3];     ! [27] the group name of the
                              !      user being added.
  INT     username[0:3];      ! [31] the user name of the user
                              !      being added.
  INT     group^id;           ! [35] the group number of the
                              !      user being added.
  INT     user^id;            ! [36] the user number of the
                              !      user being added.
  END;
```

The form of the reply to the add user message is:

```
STRUCT  adduser^reply;
  BEGIN
  INT     replycode;          ! [0] 0=allow addition of user,
                              !     1=disallow addition of
                              !       user.
  STRING  replytext[0:n];     ! [1] optional message to be
                              !     printed.  Maximum of 132
                              !     bytes.
  END;
```

The delete user message is sent whenever the user attempts to delete a user from the system. The $CMON reply indicates whether the user should be deleted, and contains an optional display message. If the $CMON process is not running, then no $CMON delete user restrictions will be in effect. Any user can be deleted from the group if the current user is the group manager. Note that a super ID can delete any user.

The form of the delete user message is:

```
STRUCT  deluser^msg;
  BEGIN
  INT     msgcode;              ! [0] -55
  INT     userid;               ! [1] user ID of user deleting
                                !     the user.
  INT     cipri;                ! [2] initial priority of CI.
  INT     ciinfile[0:11];       ! [3] name of CI's command file.
  INT     cioutfile[0:11];      ! [15] name of CI's list file.
  INT     groupname[0:3];       ! [27] the group name of the
                                !      user being deleted.
  INT     username[0:3];        ! [31] the user name of the user
                                !      being deleted.
  END;
```

The form of the reply to the delete user message is:

```
STRUCT  deluser^reply;
  BEGIN
  INT     replycode;            ! [0] 0=allow deletion of user,
                                !     1=disallow deletion of
                                !       user.
  STRING  replytext[0:n];       ! [1] optional message to be
                                !     printed.  Maximum of 132
                                !     bytes.
  END;
```

The alter priority message is sent whenever the user attempts to alter the priority of a process. The $CMON reply indicates whether the process' priority should be changed and contains an optional display message. If the $CMON process is not running, then there are no $CMON alter priority restrictions. The priority of any process which has the same process access ID as the user can be changed. Note that the super ID can change the priority of any process.

The form of the alter priority message is:

```
STRUCT  altpri^msg;
  BEGIN
  INT    msgcode;              ! [0] -56
  INT    userid;               ! [1] user ID of user altering
                               !     priorities.
  INT    cipri;                ! [2] initial priority of CI.
  INT    ciinfile[0:11];       ! [3] name of CI's command file.
  INT    cioutfile[0:11];      ! [15] name of CI's list file.
  INT    crtpid[0:3];          ! [27] process ID of the process
                               !      whose priority is to be
                               !      altered.
  INT    progname[0:11];       ! [31] expanded program file
                               !      name of the process whose
                               !      priority is to be altered.
  INT    priority;             ! [43] the new priority.

  END;
```

The form of the reply to the alter priority message is:

```
STRUCT  altpri^reply;
  BEGIN
  INT     replycode;           ! [0] 0=allow priority to be
                               !     altered,
                               !     1=disallow priority to be
                               !       altered.
  STRING  replytext[0:n];      ! [1] optional message to be
                               !     printed.  Maximum of 132
                               !     bytes.
  END;
```

The password message is sent whenever the user attempts to change his or her password. The $CMON reply indicates whether the user's password can be changed, and contains an optional display message. If the $CMON process is not running, then there are no $CMON password restrictions. The user is allowed to change his or her password at any time.

The form of the password message is:

```
STRUCT  password^msg;
  BEGIN
  INT     msgcode;             ! [0] -57
  INT     userid;              ! [1] user ID of user changing
                               !     the password.
  INT     cipri;               ! [2] initial priority of CI.
  INT     ciinfile[0:11];      ! [3] name of CI's command file.
  INT     cioutfile[0:11];     ! [15] name of CI's list file.
  END;
```

The form of the reply to the password message is:

```
STRUCT  password^reply;
  BEGIN
  INT     replycode;          ! [0] 0=allow password to be
                              !     changed,
                              !     1=disallow password to be
                              !       changed.
  STRING  replytext[0:n];     ! [1] optional message to be
                              !     printed.  Maximum of 132
                              !     bytes.
  END;
```

The remote password message is sent whenever the user attempts to change his or her remote password. The $CMON reply indicates whether the user's remote password can be changed, and contains an optional display message. If the $CMON process is not running, then there are no $CMON remote password restrictions. The user is allowed to change his or her remote password at any time.

The form of the remote password message is:

```
STRUCT  remotepassword^msg;
  BEGIN
  INT     msgcode;            ! [0] -58
  INT     userid;             ! [1] user ID of user changing
                              !     remotepasswords.
  INT     cipri;              ! [2] initial priority of CI.
  INT     ciinfile[0:11];     ! [3] name of CI's command file.
  INT     cioutfile[0:11];    ! [15] name of CI's list file.
  INT     sysname[0:3];       ! [27] change the remotepassword
                              !      for this system. "*"
                              !      indicates all systems.
  END;
```

The form of the reply to the remote password message is:

```
STRUCT  remotepassword^reply;
  BEGIN
  INT     replycode;          ! [0] 0=allow the remotepassword
                              !     to be changed.
                              !     1=disallow the
                              !       remotepassword to be
                              !       changed.
  STRING  replytext[0:n];     ! [1] optional message to be
                              !     printed.  Maximum of 132
                              !     bytes.

  END;
```

SECTION 6

INTERFACING TO TERMINALS

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). Refer to your System Management Manual.

The file system provides for data transfers between application processes and terminals in blocks of 0 to 4,095 bytes. Page mode (also called block mode) limits the size of a block to 256 bytes, excluding certain control characters.

NOTE

This section provides an overview of the terminal process, known as TERMPROCESS, from the operating system point of view. It should not be construed as the final authority on any terminal interface. Refer to the applicable terminal manuals for detailed information on their use. See the list of related publications in the front of this manual, and refer to the Guide to Software Manuals for a complete list of all current Tandem software publications.

One manual particularly recommended for your perusal is the 653X Multi-Page Terminal Programmer's Guide.

GENERAL CHARACTERISTICS OF TERMINALS

Terminals are accessed either by their $<device-name> or $<logical-device-number>.

Multiple concurrent opens are permitted for a given terminal.

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

The terminal device type is 6 for both conversational-mode and page-mode terminals.

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).

There are four system-defined interrupt characters:

Backspace, %10
Line cancel, %30
End of file (CTRL Y), %31
Carriage return, %15

Backspace and line cancel are not seen by application processes. End of file and carriage return are seen by application processes. They are indicated to application processes by the completion of a read to a terminal.

Application processes can programmatically specify nonsystem-defined interrupt characters by using the SETMODE procedure. Receipt of an application-defined interrupt character is always indicated to the application process. The SETMODE functions relevant to this section are presented at the end of this section. The SETMODE procedure is documented in the System Procedure Calls Reference Manual.

- Transparency mode: Whether or not the asynchronous multiplexer hardware is to check for the interrupt characters is configured at system generation time. The configured mode can be overridden through a call to SETMODE.

- Echo mode: Whether or not the asynchronous multiplexer hardware is to echo incoming characters back to the terminal is configured at system generation time. The configured mode can be overridden through a call to SETMODE. The no-echo mode can be useful for entering user-defined security passwords at a terminal.

Default transfer mode (conversational mode or page mode) for a device is configured at system generation time, but can be overridden through a call to the SETMODE procedure.

Parity generation mode (odd, even, or none) for a terminal can be set at SYSGEN and, if necessary, modified later by SETMODE. Parity checking can also be enabled or disabled through use of a SETMODE function.

BREAK (or ATTENTION) signalling from terminals can be handled through special SETMODE functions.

Application checksum processing: a terminal can be configured so that either one or two characters following a terminating ETX character are sent to the application program.

Device-dependent functions are configured at system generation time--refer to the System Management Manual. The following selections must be determined by the SYSGEN:

Baud rate
Character size
Terminal and system parity generation
Connection type (hard-wired, modem, echo)
Enable or disable checking for interrupt characters (transparency)
Half-duplex modem turnaround characters
Read Completion on ETX (for checksum processing)
Default transfer mode (conversational or page)
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 software programs using terminals open them with share access.

Default file-system spacing mode is postspace (space after printing). The spacing mode can be set to prespace (space before printing) by 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 and 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 and 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 (prespacing and postspacing)

SETMODENOWAIT	is used in the same way as SETMODE except in a nowait manner in an open file.
CLOSE	stops access to an open file.
SETPARAM	See the appropriate Data Communication Manual for information on SETPARAM and "BREAK Handling".

ACCESSING TERMINALS

As with any other file, you should access a terminal through the OPEN procedure. For example, to access the home terminal associated with the running of an application process, you can include the following in your program:

```
INT .HOME^TERM [0:11],                  ! data declarations.
    HOME^TERM^NUM,                      !
   .TERM^BUFFER[0:35];                  !
  .
CALL MYTERM ( HOME^TERM );
  .
```

MYTERM is a process control procedure that returns the file name of the home terminal.

```
  .
CALL OPEN ( HOME^TERM, HOME^TERM^NUM );
  .
```

OPEN returns a file number for use in calls to other file-system procedures to access the terminal.

Then to write on the terminal:

```
  .
CALL WRITE ( HOME^TERM^NUM, TERM^BUFFER, 72, NUM^WRITTEN );
  .
```

72 characters of TERM^BUFFER are written on the terminal; the value 72 is returned in NUM^WRITTEN.

To read from a terminal:

```
  .
CALL READ ( HOME^TERM^NUM, TERM^BUFFER, 72, NUM^READ );
  .
```

In this case, assuming I/O with wait, the application process is suspended until a line is input on the terminal--an indefinite period of time. The maximum number of characters permitted is 72. The actual number of characters read is returned in NUM^READ.

The WRITEREAD procedure is provided to ensure that the computer system is ready to receive data from a terminal immediately after a message is written to a terminal. This is quite useful when conversationally prompting a terminal user for input.

For example, to prompt a terminal user with a colon (:) and then wait for input, you can use the following in your application program (note that 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 a colon (:) on the terminal, then waits for input.

Note that 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 lowercase letter "a", followed by a device-control-1 character), you could include the following 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" later in this section). 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.

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

- <read-count> characters are input (regardless of interrupt character checking or transfer mode). On the return from READ or WRITEREAD, <buffer> contains <read-count> characters, <count-read> = <read-count>, and the condition code indicator is set to CCE.

- The BREAK key is pressed and break is enabled for the terminal (regardless of interrupt character checking or transfer mode). On the return from READ or WRITEREAD, nothing is transferred into <buffer>, <count-read> = 0, and the condition code is set to CCL.

- "Read termination on ETX" is enabled and an ETX character followed by the designated number of checksum characters (one or two) is input. On the return from READ or WRITEREAD, <buffer> contains <count-read> characters, the last two or three characters being the ETX character and the one or two checksum characters, and the condition code indicator is set to CCE.

TRANSFER MODES

The file system supports transfers from both conversational-mode and page-mode (normal and pseudopolled) terminals.

NOTE

The principal difference between conversational mode and page mode, as far as the file system is concerned, is the action that the file system takes when a read terminates: for conversational mode, a character sequence (dependent on what terminates the read) may be sent to the terminal to control line spacing; for page mode, no such character sequence is sent.

Terminals operating in conversational mode transfer each character, as typed, to the computer system. A file transfer is terminated when a line-termination character (typically a carriage return character) is received by the computer system.

Terminals operating in page mode store each character, as typed, in an internal buffer. The entire block of characters is transferred to the computer system in one continuous stream. The transfer is usually initiated when the terminal operator presses the ENTER (or SEND or TRANSMIT) key. A file transfer is terminated when a page termination character (typically a carriage return or ETX character) is received by the computer system.

Figure 6-1 illustrates the operation of transfer and page modes.

Normal Page Mode Versus Pseudopolled Page Mode

Terminals operating in normal page mode transfer their block of information immediately after the terminal operator presses the ENTER (or similar) key.

Terminals operating in pseudopolled page mode transfer a control character to the computer when the terminal operator presses the ENTER (or similar) key. This character informs the computer that the terminal is ready to send a block of information. The computer responds by sending a "trigger" character back to the terminal. The terminal responds to the trigger by sending the block of information to the computer.

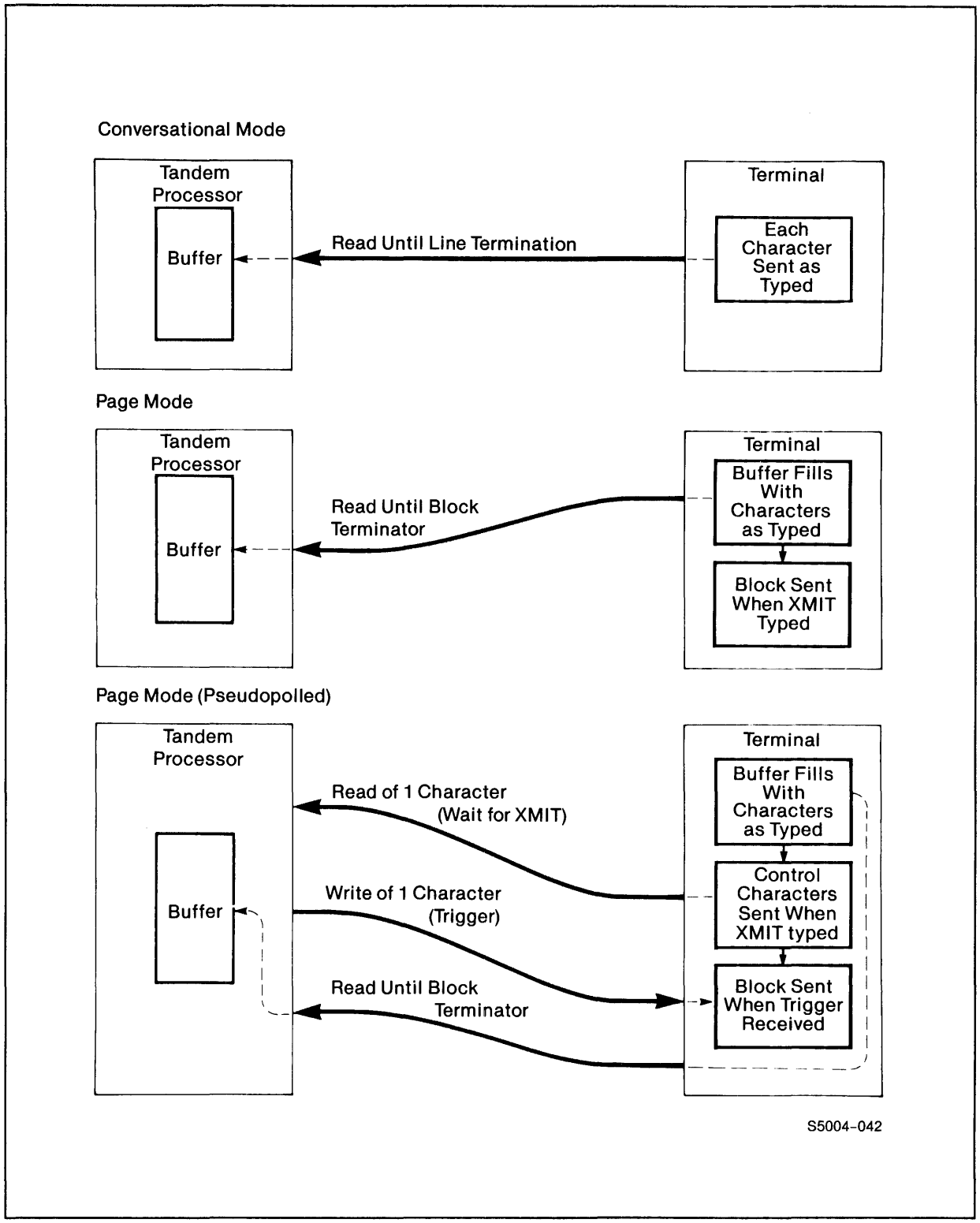


Figure 6-1. Transfer Modes for Terminals

Conversational Mode

Some characteristics of conversational-mode terminals are:

- A line-termination character (typically a carriage return) is configured for each device at system generation time.
- Initially, all four conversational-mode interrupt characters are set to system-defined values (one is the line-termination character). Application-dependent interrupt characters can be specified through the SETMODE procedure.
- The EOF indication is CTRL Y (%031).
- Procedures are available for controlling forms (or cursor) movement:

  --Vertical tab, form feed (CONTROL operation)

  --Single space (file system appends CR/LF when writing to a terminal) or no-space (SETMODE)

  --Enable or disable automatic line feed after carriage return line-termination received from terminal (SETMODE)

- The file system automatically appends a CR/LF sequence when writing to a terminal (unless no-space has been enabled through the SETMODE procedure).
- A write of zero characters results in a blank line if single spacing is enabled.
- If necessary, the file system defers subsequent access to the same terminal after issuing an automatic CR/LF sequence or when performing forms control through the CONTROL procedure. The period for each type of delay is configured at system generation time.

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 (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 CR/LF sequence to the terminal.

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

The following examples illustrate these special characteristics:

CR 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, the operator only presses RETURN:

```
<cr>
↑
|  - (initial cursor position)
```

the contents of BUFFER remain unchanged, zero is returned in NUM^READ, and the system issues a line feed to the terminal.

To terminate a line by using <read-count>:

```
.
CALL READ ( HOME^TERM^NUM, BUFFER, 10, NUM^READ );
.
```

When the terminal operator types in the 10 characters:

```
NOW IS THE
↑
|  - (initial cursor position)
```

"NOW IS THE" is returned in BUFFER and 10 is returned in NUM^READ.

```
NOW IS THE
         ↑
         |  - (final cursor position)
```

Conversational-Mode Interrupt Characters

Four system-defined characters cause special file-system action when encountered as interrupt characters: backspace, line cancel, end of file, and the configured line termination character. These characters are also the initial settings for interrupt characters.

NOTE

The default interrupt characters apply to a terminal when first opened (if configured as a conversational-mode terminal) and are restored to the initial values when dynamically changing from page to conversational-mode (specifed by SETMODE function 9--see the summary at the end of this section).

The file-system action that occurs when the line-termination character is encountered is described above. The action that occurs for the three other system-defined characters is as follows:

- Backspace (CTRL H, %010): The purpose of backspace is to permit a terminal operator to back up, then reenter one or more mistyped characters. The specific action involved depends on the particular terminal. Typically, on video terminals the cursor is backspaced one position for each backspace received. On hard-copy devices that backspace, a line feed and a backspace are issued for the first backspace received, and a single backspace is issued for each subsequent backspace received. On hard-copy devices that do not backspace, a backslash "\" is printed for each backspace entered. This operation is invisible to the application program.

- Line Cancel (CTRL X, %030): The purpose of line cancel is to permit a terminal operator to cancel out, then reenter, the current line. When line cancel is received, the file system writes a "@<cr><lf>" character sequence on the terminal. This action is invisible to the application program.

- End of File (CTRL Y, %031): The purpose of EOF is to permit a terminal operator to signal an application process that no more data will be entered. When the EOF character is received, the current file operation is considered complete; no data is transferred into the application program's buffer area, <count-read> is returned as 0, and the condition code indicator is set to CCG. The file system writes a "EOF!<cr><lf>" character sequence on the terminal.

Figure 6-2 illustrates the conversational-mode interrupt characters.

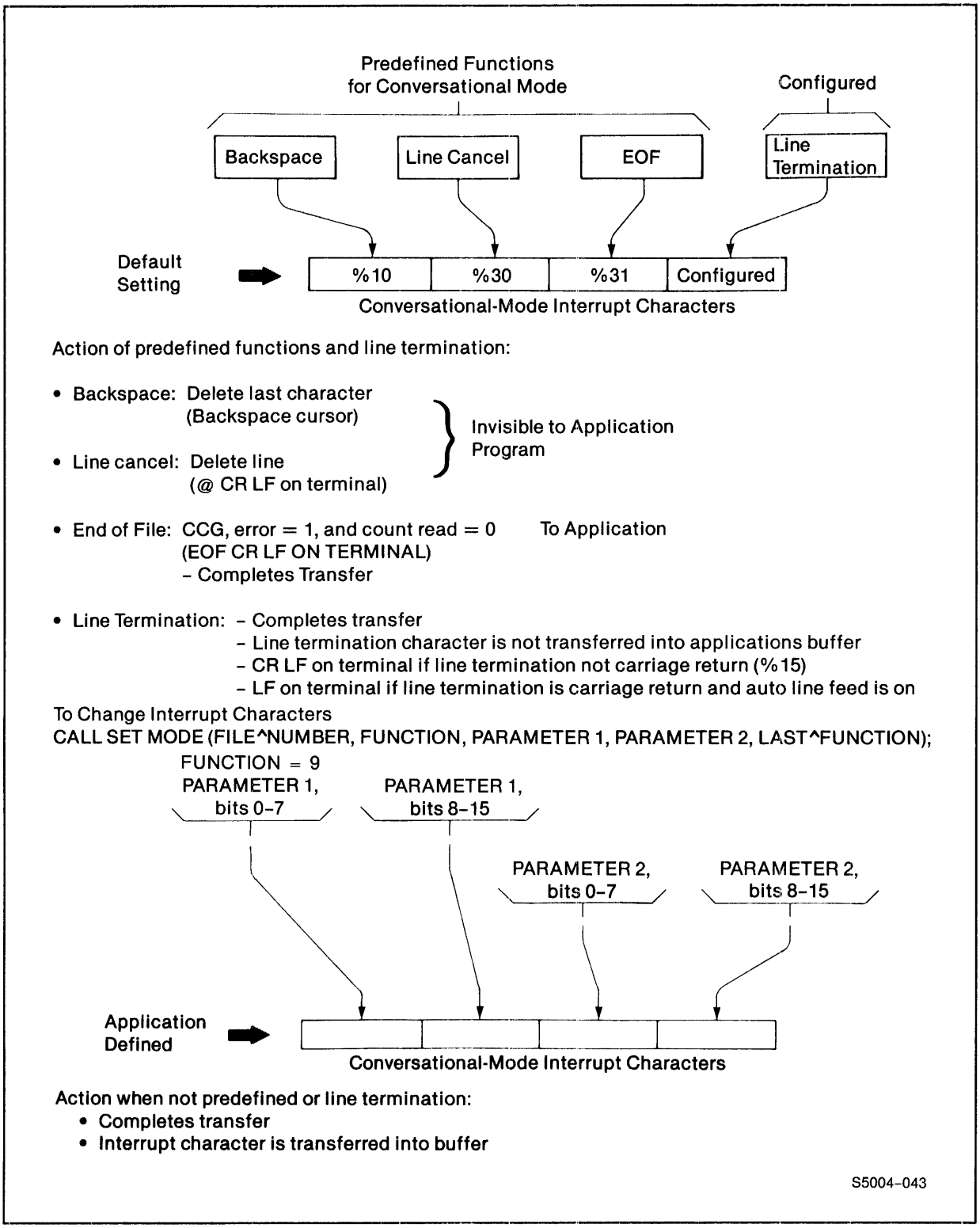


Figure 6-2. Conversational-Mode Interrupt Characters

You can change the interrupt characters for special applications by using SETMODE function 9. The file-system action when receiving any interrupt character that is not one of the system-defined values is always the same:

- The file transfer is considered complete.
- No line feed (or carriage return) is sent to the terminal; the next transfer to the terminal physically begins at the character following the interrupt character.
- The interrupt character is transferred into the application program's buffer along with the line image (if any).
- The <count-read> parameter includes the interrupt character.

Note that special application-dependent interrupt characters can be mixed with the system-defined characters.

For example, assume that a tab function is desired on a conversational-mode terminal. The end-of-file character is not required, so it is replaced by the horizontal tab character; the other system-defined characters are retained, including the line-termination character (which is configured as carriage return).

```
LITERAL SET^SIG    = 9,          ! setmode, set int. characters.
        SET^SPACE  = 6,          ! setmode, set spacing mode.
        SPACE      = 1,          ! spacing mode = single space.
        NO^SPACE   = 0;          ! spacing mode = no space.
  .
STRING INTCHARS[0:3] := [%010,    ! backspace.
                         %031,    ! line cancel.
                         %011,    ! horizontal tab.
                         %015];   ! carriage return (line
                                  ! termination).
INT IINTCHARS = INTCHARS;         ! equivalence INT to STRING.
  .
```

First a call to SETMODE sets the new 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<CTRL I>                    ! <CTRL I> is the
^                                           ! horizontal tab
|                                           ! character.
| - (initial cursor position)
```

"TODAY IS THE DAY<CTRL 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 <CTRL 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.

After the second read, information on the terminal appears as:

```
TODAY IS THE DAY_____________FOR BEGINNING
^
| - (cursor position)
```

At this point, the application process sets the terminal spacing mode to single space:

```
.
CALL SETMODE ( HOME^TERM^FNUM, SET^SPACE, SPACE );
.
```

Forms Control

The file-system SETMODE and CONTROL procedures provide you with the capability to explicitly control forms movement.

SETMODE function 6 can programmatically change between single spacing (where the file system appends CR/LF) and no spacing when writing.

For example, to place a terminal in a mode so that no line spacing occurs after writing (position the cursor following the last character written), make the following call to SETMODE:

```
LITERAL SET^SPACE = 6,          ! setmode, set spacing mode.
        SPACE     = 1,          ! spacing mode = single space.
        NO^SPACE  = 0;          ! spacing mode = no space.
.
CALL SETMODE ( HOME^TERM^NUM, SET^SPACE, NO^SPACE );
                                                ! no-spacing.
.
```

An example of using no-spacing is shown in the example for implementing a tabulation scheme following Figure 6-2. Specifically, no-spacing is used when the application tabs (writes blanks) to column 30.

Another reason for using no-spacing would be if overprinting was desired on a hard-copy terminal. In this case, the application program must append a carriage return character to the data to be written:

```
.
STRING .BUFFER[0:71] := ["WHAT EVER HAPPENED TO",%015];
.

.
CALL WRITE ( HOME^TERM^NUM, BUFFER, 22, NUM^WRITTEN );
.
```

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

```
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 nowait 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, place the following call to CONTROL 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, you must include a delay 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.

Page Mode

Some characteristics of page-mode terminals:

- A page termination character (such as CR, ETX, or EOT) is configured for each page-mode terminal at system generation time.
- Initially, all four page-mode interrupt characters are the same as the configured page termination character. You can specify application-dependent interrupt characters by using the SETMODE procedure.
- Normal and pseudopolled page mode transfers are supported (configured).

Page-Termination Character

The page-termination character, when received from a terminal, signals the computer system that the current page transfer is completed. Page-termination characters for each page-mode terminal connected to the system are specified (configured) at system generation time.

There are special characteristics associated with receiving the page-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.
- The line is terminated automatically when the number of characters specified in the <read-count> parameter are input.

NOTE

The file system does not issue a carriage return/line feed sequence after receiving the page-termination character.

Page-Mode Interrupt Characters

Initially, all four page-mode interrupt characters are set to the configured page-termination character.

NOTE

The initial page-mode interrupt characters apply to a terminal when first opened (if configured as a page-mode terminal) and are restored to the initial setting when dynamically changing from conversational mode to page mode (by using the SETMODE procedure).

You can change the page-mode interrupt characters to other values by using SETMODE function 9. The file-system action when receiving an interrupt character that is not the page-termination character is:

- The file transfer is considered complete.
- The interrupt character is transferred into the application program's buffer along with the page image (if any).
- The <count-read> parameter includes the interrupt character.

Note that special application-dependent interrupt characters can be mixed with the page termination character.

Figure 6-3 illustrates the page-mode interrupt characters.

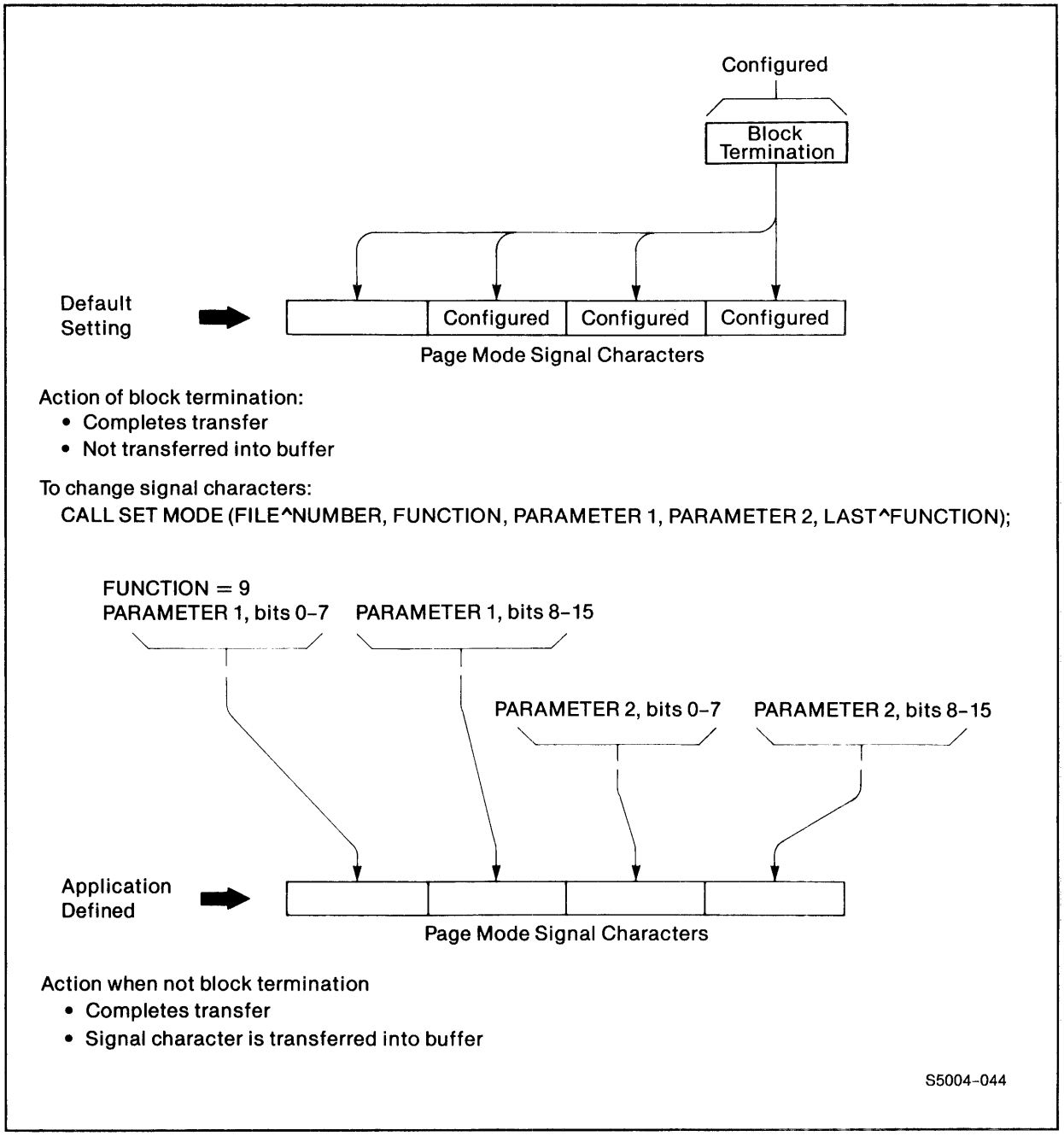


Figure 6-3. Page-Mode Interrupt Characters

The following example shows the action of the interrupt characters when dynamically changing from conversational mode to page mode, then back to conversational mode. The configured line-termination character is carriage return; the configured page-termination character is also carriage return. The terminal is configured as a conversational-mode terminal:

```
.
LITERAL CHANGE^MODE  =  8, ! for SETMODE.
        CONV^MODE    =  0,
        PAGE^MODE    =  1,
        SET^INTCHARS  =  9,
        BS^CAN  = %004030, ! backspace, line cancel.
        HT^CR   = %004415, ! horizontal tab, carriage return.
        ETX^EOT = %001404; ! end of text, end of transmission.
  .
```

The terminal is opened:

```
.
CALL OPEN ( HOME^TERM, HOME^TERM^NUM, ...);
  .
```

At this point, the interrupt characters are set to their initial conversational mode values: backspace, line cancel, end of file, and carriage return (line-termination).

Then a call to SETMODE changes the interrupt characters:

```
.
CALL SETMODE ( HOME^TERM^NUM, SET^INTCHARS, BS^CAN, HT^CR );
  .
```

At this point the interrupt characters are set to: backspace, line cancel, horizontal tab, and carriage return.

The file-system transfer mode is set to page mode for the terminal:

```
.
CALL SETMODE ( HOME^TERM^NUM, CHANGE^MODE, PAGE^MODE );
  .
```

At this point all four interrupt characters are end-of-text characters.

Then new page-mode interrupt characters are set:

```
.
CALL SETMODE ( HOME^TERM^NUM, SET^INTCHARS, ETX^EOT, ETX^EOT);
  .
```

After the call to SETMODE, the interrupt characters are end of text and end of transmission.

The file-system transfer mode for the terminal then returns 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 fewer than four are needed, some characters must be duplicated.

Pseudopolled Terminals

During system generation, each pseudopolled 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, specify a trigger character with a value of zero (null):

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 of 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 presses 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:

Your application program issues a read of one character to the pseudopolled 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 presses the ENTER key. Pressing ENTER causes a ready character (for example, device-control-2) to be sent to the computer system (causing the READ to complete).

The application then issues the trigger character (for example, device-control-1) to the terminal and issues a read of 600 characters (to ensure that the computer is ready to start reading when the terminal starts transmitting, both operations must be combined into one using WRITEREAD):

```
.
BUFFER := %010400; ! device-control-1.
.
CALL WRITEREAD ( HOME^TERM^NUM, BUFFER, 1, 600, NUM^READ );
  .
```

Calling WRITEREAD causes 300 words (600 bytes) of system buffer space to be allocated. The device-control-1 character is sent to the terminal, causing it to send a page of information back to the computer system. The page is returned to the application process in BUFFER, and the actual number of bytes read is returned in NUM^READ. (As with any file-system operation, the system buffer space is deallocated after the read completes.)

Simulation of Pseudopolling

Terminals without pseudopolling (such as P/N 6511 and P/N 6512), can simulate this feature by having the terminal operator signal that data is being sent by pressing a function key (rather than the SEND PAGE key). Upon receipt of the characters transmitted because of the function key, your application program issues a SEND PAGE command to the terminal to have the data returned (through a call to the file-system WRITEREAD procedure).

The following is an example of simulated psuedopolling coding with the 6511 or 6512 terminal.

In this example, terminal operation is controlled by sending a series of escape-sequence functions to the terminal (for definition of the escape-sequence functions for 6511 and 6512 terminals, refer to the Lear-Siegler ADM-2 Operator's Handbook).

```
INT .BUFFER[0:1023],   ! screen input buffer.
    .CONTROL^BUF[0:1], ! for sending control sequences to
                       ! terminal.
     COUNT^READ;       ! number of bytes read.

STRING
    .SBUFFER      := @BUFFER '<<' 1,      ! redefine as byte
                                          ! array.
    .SCONTROL^BUF := @CONTROL^BUF '<<' 1; !

LITERAL
  READCOUNT    = 2048, ! read count for full screen.
  ESC          =  %33, ! ASCII ESC character.
  CLEAR^SPACES =  ";", ! "esc,clear^spaces"= clear screen.
  TBLOCK^MODE  =  "B", ! "esc,tblock^mode"= set tblock mode.
  TCONV^MODE   =  "C", ! "esc,tconv^mode" = set tconv. mode.
  SENDALL      =  "5", ! "esc,sendall"    = send page
                       ! unprotected, to cursor.
  CHANGE^MODE  =    8, ! set mode, set transfer mode function.
  CONV^MODE    =    0, ! set mode param 1, set conv. 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 pressed.
SCONTROL^BUF ':=' [ESC, CLEAR^SPACES, ESC, TBLOCK^MODE];
CALL WRITEREAD ( TERM^NUM, CONTROL^BUF, 4, 2 );
IF = THEN       ! function key pressed.
  BEGIN
     .
You can examine the returned data in the buffer to
determine which function key was pressed.
     .
  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 pressed.

```
! 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.
```

This WRITE is issued to the terminal to put the terminal into conversational mode (ESC "C").

TRANSPARENCY MODE (INTERRUPT-CHARACTER CHECKING DISABLED)

You can disable the interrupt character checking for a terminal through a call to SETMODE function 14. If interrupt character checking is disabled, a READ or WRITEREAD terminates only when the number of bytes specified by the <read-count> parameter have been read or, if break is enabled for the terminal, when the BREAK key is pressed.

CHECKSUM PROCESSING (READ TERMINATION ON ETX CHARACTER)

To permit an application to perform checksum processing, use a terminal READ or WRITEREAD to terminate on either the first or second character following an ETX (end of text) character. Use SETMODE function 13.

If you specify read termination on ETX, the ETX character and the next one or two (checksum) characters are returned in <buffer> and reflected in the <count-read> value of the call to READ or WRITEREAD.

NOTE

Interrupt character checking is not affected by the read termination on ETX character mode.

ECHO

The file-system terminal echo mode is configured for each terminal at system generation time. If echo is specified and the file-system terminal-transfer mode is conversational, each character as received from the terminal is sent back to the terminal, where it is displayed (if the physical terminal is in full-duplex mode).

The configured echo mode can be changed programatically using SETMODE function 20. The <parameter-1> values for this function are:

0 = system does not echo
1 = system echoes

You can use the no-echo mode when entering an application-defined password at a terminal. If the terminal is in full-duplex mode and the file-system terminal echo mode is "system does not echo", then a password will not be displayed. Following entry of the password, the echo mode would be set to "system echoes".

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 nowait 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 reprompting for the account number. To reprompt, include code similar to the following in your application program:

```
      .
DEFINE FIVE^MINUTES = 30000D#;
LITERAL TIMEOUT = 40;
INT ERROR, .BUFFER[0:599];
      .
      .
REPROMPT:
      .
   BUFFER ':=' "PLEASE ENTER ACCOUNT NUMBER";
   CALL WRITEREAD ( TERM^NUM, BUFFER, 27, 400, NUM^READ );
   IF < THEN ...  ! checks if WRITEREAD was successful
   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 REPROMPT
       ELSE .....;
     END;
      .
```

The message "PLEASE ENTER ACCOUNT NUMBER" is issued every 5 minutes until the operator responds.

If the call to AWAITIO had been for any file (for example, <filenum> = -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 your application program. However, you 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.

You can use a CONTROL operation to signal your application process that a communication link (signalled by an incoming call) is established. Selected CONTROL operations are included in the table at the end of this section. All of the CONTROL operations are described in the System Procedure Calls Reference Manual.

For example, an application process wants to accept an incoming call from a modem designated "$LINE1". However, the process must continue with other processing functions. You could use the following:

```
  .
LITERAL NO^WAIT = 1, WAIT^ON^CALL = 11,  ! data declarations.
        TIMEOUT = 40;                    !
  .
INT .LINE1[0:11] := ["$LINE1", 9 * ["  "]];
INT LINE1^FNUM, CALL^RECEIVED := 0;
  .
```

Then the modem is opened, allowing one concurrent I/O operation:

```
  .
CALL OPEN ( LINE1, LINE1^FNUM, NOWAIT );
  .
```

The file system enables the modem for an incoming call. Your application process continues executing.

```
  .
```

Then a call to CONTROL that specifies "wait for modem connect" is made.

```
  .
CALL CONTROL ( LINE1^FNUM, WAIT^ON^CALL );
```

Periodically, a "check for completion" call to the AWAITIO procedure is made to determine if a call has been received:

```
LOOP:
IF NOT CALL^RECEIVED THEN
  BEGIN
    CALL AWAITIO ( LINE1^FNUM, ...., 0D );
    IF = THEN CALL^RECEIVED := 1
    ELSE
      BEGIN
        CALL FILEINFO ( LINE1^FNUM, ERROR );
        IF ERROR <> TIMEOUT THEN ...; ! trouble.
      END;
  END;
```

CONTROL operation 12 allows the modem to be disconnected without having to close the associated file.

BREAK FEATURE

The file system includes special features that permit a terminal operator to signal a process by pressing the BREAK key. An example of BREAK usage is when running an application program through the command interpreter process; pressing 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 by using a SETMODE function (the process that has BREAK enabled is referred to as the "owner" of BREAK).
- You can enable BREAK for only one process at a time.
- If the terminal is opened by the backup process of a process pair (for example by using CHECKOPEN by primary or backup open by backup), the backup process automatically becomes the owner of BREAK if its primary failed while owning BREAK. Refer to Section 12 for more information on checkpointing.
- When BREAK is pressed at a terminal, 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 pressed.
- The terminal where BREAK was pressed 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 pressed, it is disabled, and further breaks on that terminal are ignored. BREAK is automatically reenabled for the owner when a READ or WRITEREAD procedure is executed to the terminal.
- After pressing BREAK, an application not wishing to issue a READ or WRITEREAD to a terminal reenables BREAK using 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 pressed 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 pressed.

BREAK System Message

The form of the BREAK system message is:

<sysmsg>	= -20
<sysmsg>[1]	= logical device number, in binary, of device where BREAK was pressed.
<sysmsg>[2]	= system number, in binary, of device where BREAK was pressed.

This message is received by a process through its $RECEIVE file if break monitoring is specified (through a call to SETMODE) and BREAK is pressed on a terminal being monitored.

Using BREAK (Single Process per Terminal)

To use the break feature, you must indicate in your application process the terminals to be monitored for a BREAK signal and the process ID of the process to receive the system BREAK message.

The SETMODE function to enable this type of break monitoring is:

<function> = 11, set BREAK ownership and terminal access mode after break

<parameter-1> = <cpu,pin>, enable BREAK for process <cpu,pin>

<parameter-2> = 0

The SETMODE function to disable this type of break monitoring is:

<function> = 11, set BREAK ownership

<parameter-1> = 0, disable BREAK

<parameter-2> = 0

For example, to arm BREAK, you could place the following in your application program:

```
LITERAL SET^BREAK^OWNER = 11,
        NORMAL^MODE     =  0;
  .
INT  BREAK^RECEIVED := 0,
    .RECV^BUF[0:66];
  .
```

Then BREAK is enabled by calling SETMODE function 11 and passing the file number of the terminal where BREAK is to be enabled and the <cpu,pin> of the owner:

```
.
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 pressed, a system BREAK message is sent to this process.

A read is issued to the $RECEIVE file (open as a nowait 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 pressed. 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 );
```

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 6-4 illustrates the break sequence when a terminal is controlled by a single process.

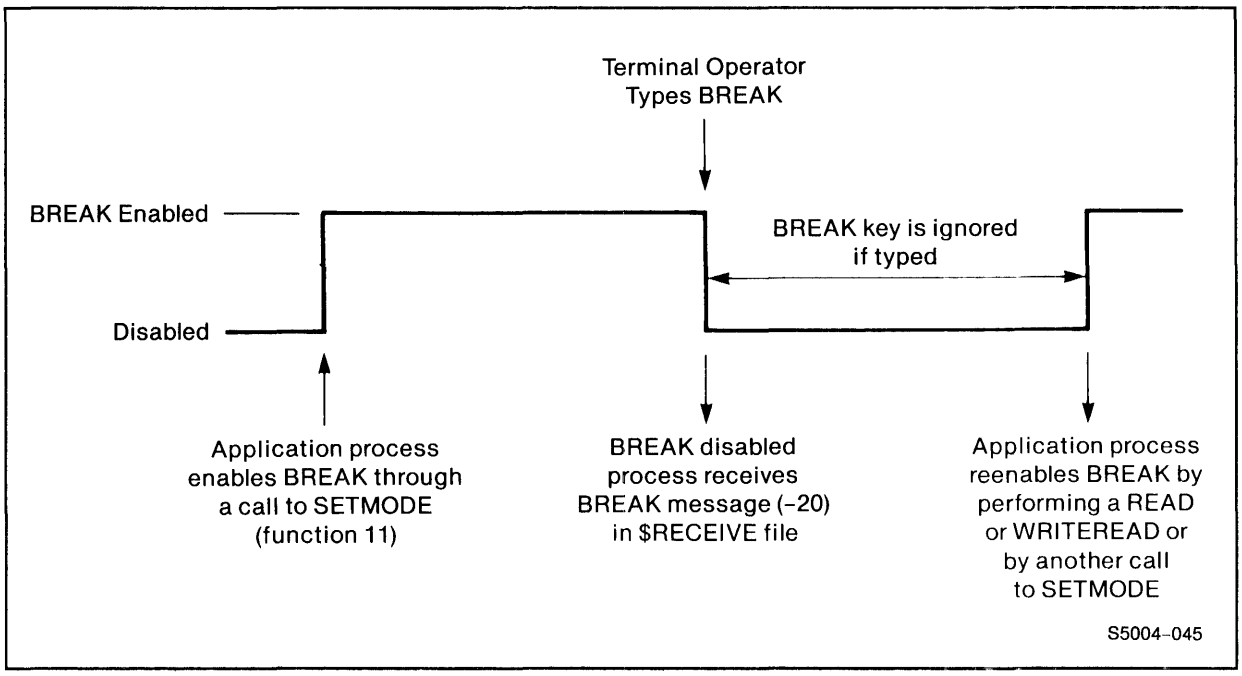


Figure 6-4. Break: Single Process per Terminal

Using BREAK (Multiple Processes per Terminal)

Keep in mind that when more than one process is accessing the same terminal, only the last process to arm BREAK receives the BREAK system message. Therefore, it is important that each process in such an environment keep track of the previous process that enabled BREAK and restore BREAK to that process when finished.

The SETMODE function to perform this type of break monitoring is:

<function> = 11, set BREAK ownership.

<parameter-1> = <cpu,pin>, enable BREAK for process <cpu,pin>.

<parameter-2> = 0.

<last-params> = last owner and last terminal access mode on return from SETMODE.

The SETMODE function to return ownership to previous BREAK owner is:

<function> = 11, set BREAK ownership.

<parameter-1> = last owner.

<parameter-2> = last terminal access mode.

<last-params> = last owner and last terminal access mode on return from SETMODE.

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, 0 is returned to LAST^OWNER. BREAK is now enabled for this process (that is, the process will receive the BREAK message if the BREAK key is pressed).

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 reenables 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 pressed, the command interpreter receives 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 pressed.

This is done in three steps:

1. First, when BREAK is enabled, break mode is specified. This means that, after the BREAK key is pressed, the terminal is put in break mode, and only file operations having break access are permitted with the terminal.

   The SETMODE function to perform this type of break monitoring is:

   <function> = 11, set BREAK ownership and terminal access mode after break.

   <parameter-1> = <cpu,pin>, enable BREAK for process <cpu,pin>.

   <parameter-2> = 1, terminal access mode after break = break mode.

   <last-params> = last owner and last terminal access mode on return from SETMODE.

2. After a BREAK message has been received, the application process, through a call to SETMODE, specifies that subsequent file operations to the terminal have break access. The application process can then communicate with the terminal in the usual manner. (Attempts by other processes with normal access will be rejected with an error indication.)

   The SETMODE function to specify the file access mode for this type of break handling is:

   <function> = 12, set file access type.

   <parameter-1> is omitted.

   <parameter-2> = 1, file access mode = break access.

3. When finished processing the break, the application process, through another call to SETMODE, returns the file access type to normal access and the terminal access mode to normal mode (permitting any type access to the terminal).

   The SETMODE function to restore the terminal access mode to normal mode, and file access type to normal access, 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.

Steps 2 and 3 are repeated for each BREAK message received.

4. When the process finishes monitoring BREAK, BREAK is returned to the previous owner by means of SETMODE function 11, as described above in "Using BREAK (Multiple Processes per Terminal)". Note that the previous owner is characterized by an internal number, rather than by the owner's <cpu, pin>.

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.

Unless more than one process is accessing a terminal, normal access (<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 message received.
ELSE
IF > THEN                    ! system message 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 nonbreak operations to the terminal indicated by HOME^TERM^NUM are rejected.

However, this process, because break access was specified, can access the terminal.

```
  .
  .
```

When finished processing the break, the application process permits normal access to the terminal:

```
  .
CALL SETMODE ( HOME^TERM^NUM, SET^ACCESS,
               NORMAL^MODE, NORMAL^ACCESS );
  .
    END
  ELSE
  IF BUFFER = ... THEN    ! some other system message.
END
ELSE ... ;                ! error
```

Figure 6-5 illustrates the action of break mode.

SETMODE function 12 can also be used to gain exclusive access to a terminal even though break has not been typed. This is done by specifying "terminal access mode = break mode" and "file access type = break access".

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.

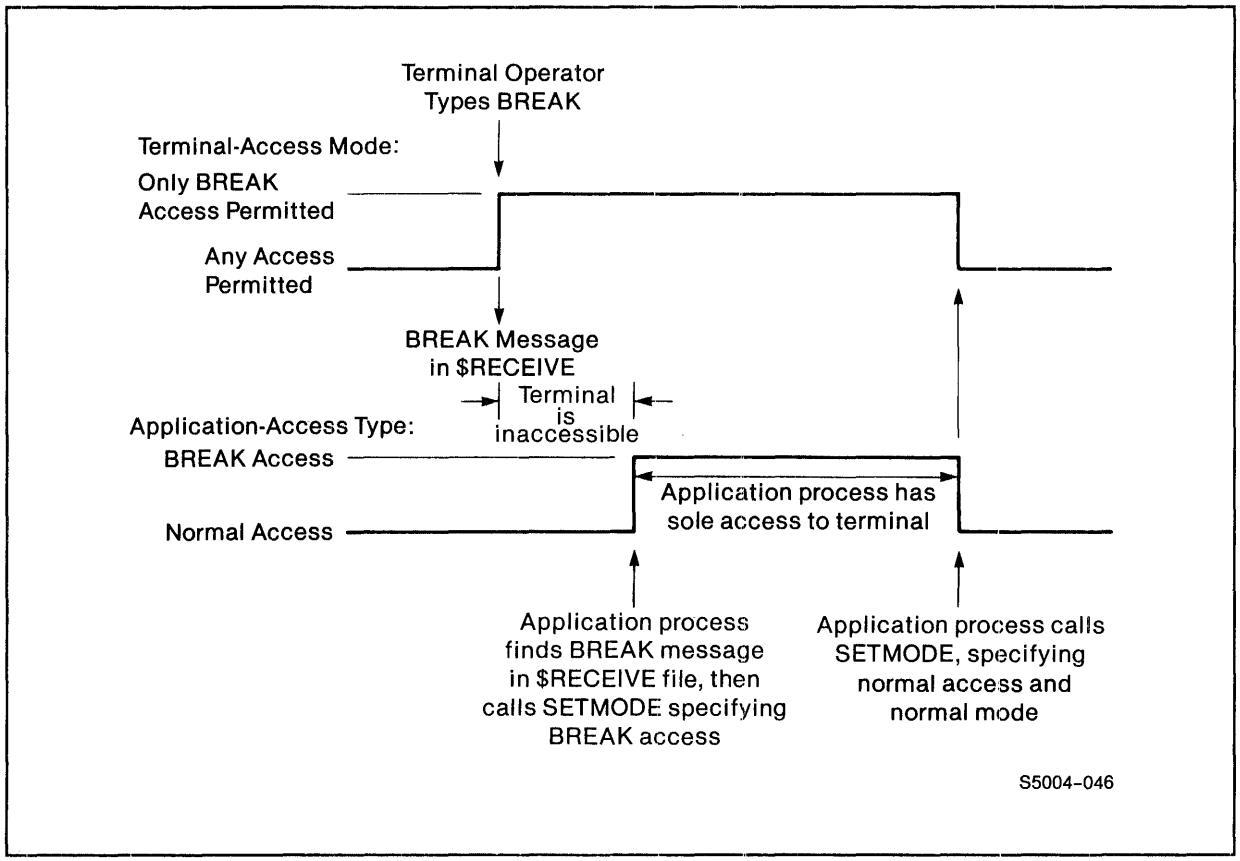


Figure 6-5. Break Mode

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 are 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 6-6.

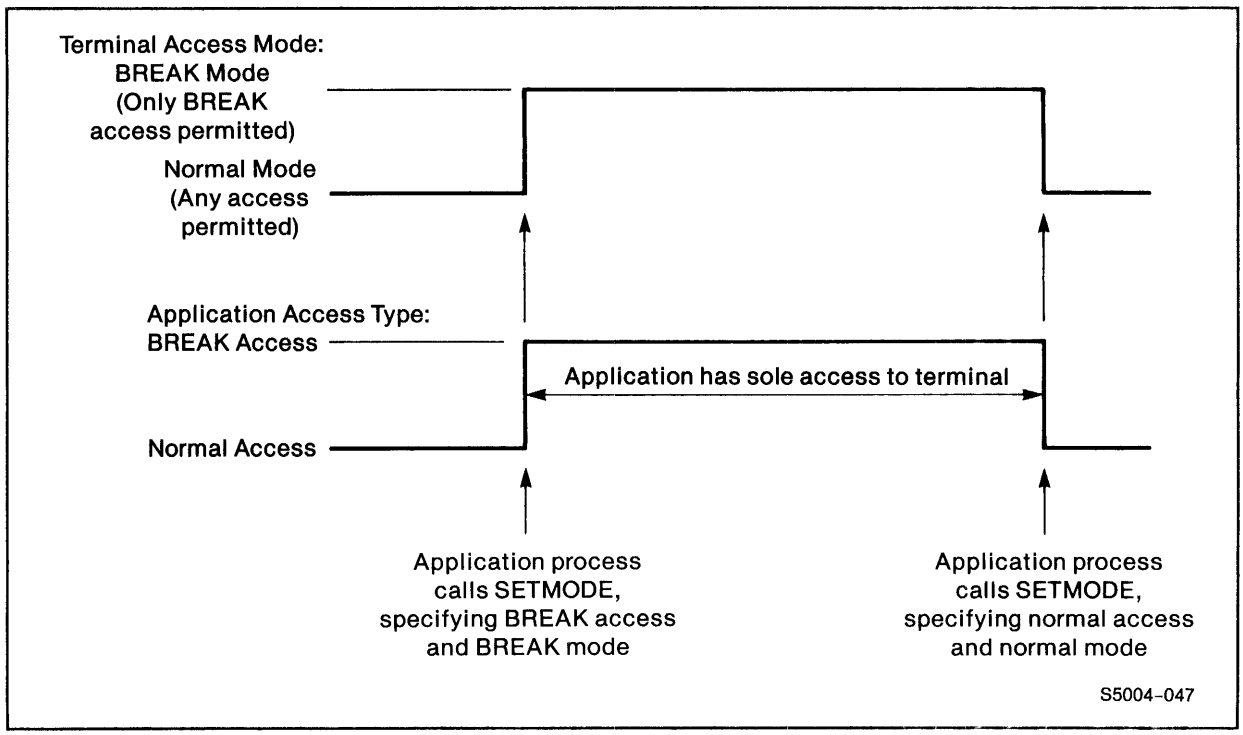


Figure 6-6. Exclusive Access Using BREAK

ERROR RECOVERY

Error recovery for terminals, in most cases, simply consists of retrying the current operation. Certain errors, involving timed reads and the break feature, require consideration. You can obtain a brief description of most errors by entering the COMINT command ERROR with the error number. All errors are fully described in the System Messages Manual.

Operation Timed Out (Error 40)

This error indicates that the terminal operator did not complete a data entry within the time allotted by a call to AWAITIO. Any data entered before the timeout occurred is lost. Therefore, a message should be sent telling the terminal operator to reenter the last data.

BREAK (Errors 110 and 111)

Pressing BREAK on a terminal where BREAK is enabled can cause an application process to receive either of two errors:

Error 110 (only break operations permitted)

Error 111 (operation aborted because of BREAK).

The action taken for these errors depends on whether the process receiving the error is the one with BREAK enabled (the process that receives the BREAK message).

Error 110 indicates that BREAK was pressed 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.

Error 110 implies that no data was transferred.

Error 111 indicates that BREAK was pressed 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 pressed. 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 procedure DELAY.

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.

Modem Error (Error 140)

This error occurs if the carrier signal to the modem was lost. The carrier loss may be a permanent or just a momentary loss. In either case, it must be assumed that data was lost. The first

time error 140 occurs, a retry message should be sent to the terminal operator (if reading) before retrying the operation. If error 140 occurs on the retry, then the connection with the remote terminal is lost. Typical action when the second error occurs is to call the CONTROL procedure to "disconnect the modem" and follow that by a call to CONTROL to "wait for modem connect".

Path Error (Errors 200-255)

The application program should keep track of the number of times any of the path errors from 201 through 229 occurs on a particular file. The occurrence of one of these errors indicates that one path to the associated device is down. A second consecutive occurrence of one of these errors indicates that both paths to the device are down and that the device is no longer accessible. Alternatively, the process may have created another backup (because of a CPU reload or an action by the application program).

If an error 210 through 231 occurs, the operation failed at some indeterminate point. If reading, a retry message should be sent to the terminal operator before retrying the read.

SUMMARY OF 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 (hang up):

<parameter> = none

Terminal SETMODE Operations

<function>

6 = set system spacing control:

<parameter-1>.<15> = 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>.<15> = 0 LFTERM line feed from terminal or network (default)
= 1 LFSYS system provides line feed after line-termination by CR

<parameter-2> is not used.

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

     <parameter-1>.<15> = 0, conversational mode
                       = 1, page mode

     <parameter-2> sets the number of retries of I/O
     operations; <parameter-2> is used with 6520 terminals
     only.


 9 = 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 = set parity checking by system (default is configured):

     <parameter-1>.<15> = 0, no checking
                        = 1, checking

     <parameter-2> is not used.

11 = set break ownership:

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

     and terminal access mode AFTER break is pressed:

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

```
12 = set terminal access mode:

     <parameter-1>.<15> = 0, normal mode (any type file
                             access is permitted)
                       = 1, break mode (only break-type
                             file access is permitted)

     and file access type:

     <parameter-2> = 0, normal access to terminal
                   = 1, break access to terminal

13 = set system read termination on ETX character (default is
     configured):

     <parameter-1> = 0, no termination on ETX
                   = 1, termination on first character after
                        ETX
                   = 3, termination on second character after
                        ETX

     <parameter-2> is not used.

14 = set system read termination on interrupt characters
     (default is configured):

     <parameter-1> = 0, no termination on interrupt
                        characters (transparency mode)
                   = 1, termination on any interrupt
                        character input

     <parameter-2> is not used.

20 = set system echo mode (default is configured).

     <parameter-1>.<15> = 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
                    16, baud rate = 200

     <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.

24 = set parity generation by system:

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

     <parameter-2> is not used.

27 = set system spacing mode:

     <parameter-1>.<15> = 0, postspace (default setting)
                        = 1, prespace

     <parameter-2> is not used.
```

28 = reset to configured values:

<parameter-1> = 0,
<parameter-2> is not used.

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 does not move) 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 does not move) 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].

67 = AUTODCONNECT for full duplex modems: Monitor carrier detect, or data set ready

<parameter-1> = 0, disable AUTODCONNECT (default setting)
1, enable AUTODCONNECT

<parameter-2> is not used.

110 = Set SISO mode (Shift in, shift out)

<parameter-1> = 0, disable SISO
1, enable SISO

<parameter-2> is not used.

113 = Set screen size

<parameter-1> = screen width (40, 66, 80, or 132)

<parameter-2> = screen depth (25 or 28)

Only the following four screen formats are supported for 654x terminals (width x depth):

40 x 25 80 x 25 66 x 28 132 x 28

<last-params>, if present, returns the previous setting of <parameter-1> and <parameter-2>. The format is:

<last-params>[0] = old <parameter-1>
<last-params>[1] = old <parameter-2>

SECTION 7

INTERFACING TO LINE PRINTERS

This section describes the interfaces to line printers, including the use of SETMODE and CONTROL functions for controlling line printers. Device-specific information on the 5508, 5520, and 5530 printers is included.

The file system provides for data transfers from application processes to line printers in blocks of 0 characters (blank line) to the maximum number of characters permitted in one line of print.

GENERAL CHARACTERISTICS OF LINE PRINTERS

Line printers can be accessed by either $<device-name> or $<logical-device-number>.

Default file-system spacing mode is postspace (space after printing). The spacing mode can be set to prespace (space before printing) by use of a SETMODE function.

The procedures available for explicitly controlling forms movement are:

CONTROL	Skip to VFU channel or skip a number of lines.
CONTROLBUF	Load programmable VFU or direct-access vertical format unit (DAVFU) for the model 5520 printer.
SETMODE and SETMODENOWAIT	No-space or single-space after printing. Disable or 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 using a call to the CONTROL procedure.

A standard vertical format unit (VFU) tape is supplied with some line printers (see summary at the end of this section).

It is the responsibility of application processes to handle "paper out" and "not ready" conditions.

All programs supplied with the software release, when accessing a line printer, open the printer with exclusive access such as (OPEN, <flags>.<9:11> = 1).

Line printer device type is 5.

SUMMARY OF APPLICABLE PROCEDURES

The following procedures are used to perform input-output operations with line printers:

DEVICEINFO	provides the device type and configured record length for a designated line printer.
OPEN	establishes communication with a file.
WRITE	prints a line on the line printer.
CONTROL	is used for forms control.
CONTROLBUF	loads the DAVFU for the model 5520 printer.
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 and clears the auto perforation skip and single-spacing functions.
SETMODENOWAIT	is used the same as SETMODE except in a nowait manner on an open file.
CLOSE	stops access to an open file.

ACCESSING LINE PRINTERS

Like any other file, a printer is accessed through the OPEN procedure. For example, to access a printer referenced by the device name "$LP1", the following is written in the application program:

```
INT .PTR [0:11] := ["$LP1", 10 * ["  "]];
     FILE^NUM,
    .PTR^BUFFER[0:65];

LITERAL EXCL^ACC = %20;
   .
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.

If the <count-written> parameter is present in the call to WRITE, a count of the number of characters actually printed is returned.

FORMS CONTROL

The file-system CONTROL and SETMODE procedures provide 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. You can disable both of these movements using the SETMODE procedure. Any automatic forms movement always takes place after a line is printed.

The CONTROL procedure is used to advance forms by means of a format tape (VFU) installed in the printer, by means of a programmable format unit (DAVFU) or by advancing forms a specified number of lines as required. The operations performed by CONTROL depend upon the type of printer (device subtype); see the summary at the end of this section.

The file system does not automatically set the printer form to a top-of-form position at file open time. This is desirable because it may be necessary for the operator to manually install and align a special form. However, the application can cause the paper to be positioned at top of form at open time as follows:

```
LITERAL FORMS^CONT    = 1,
        TOF           = 0,
        SKIP^ONE^HALF = 5;
   .
   .
CALL OPEN ( PTR^FNAME, FILE^NUM, EXCL^ACC );
IF < THEN ... ;
CALL CONTROL ( FILE^NUM, FORMS^CONT, TOF );
   .
```

positions the form to the top-of-form position.

You can use the CONTROL procedure to advance paper to specific locations. For example, to skip to the next one-half page using the standard VFU tape, make the following call to CONTROL:

```
   .
CALL CONTROL ( FILE^NUM, FORMS^CONT, SKIP^ONE^HALF );
   .
```

advances the form one-half page.

If at some point you wish to advance the paper a specific number of lines, you can use the CONTROL procedure. For example, to advance 30 lines on a belt-type printer, use the following call to CONTROL:

```
   .
LITERAL ^30^LINES = 30 + 16;              ! data declaration.
   .
   .
CALL CONTROL ( FILE^NUM, FORMS^CONT, ^30^LINES );
   .
```

advances the form by 30 lines.

To perform single spacing with automatic perforation skip (default modes), the file system skips on channel 2 of the printer's format tape. Therefore, if single-spacing and automatic page eject are desired, the VFU tape should be punched in channel 2 for each of lines 1 through 60. The file system does not use the VFU tape when single spacing without automatic perforation skip.

Overprinting can be accomplished with SETMODE function 6. For example, to overprint a single line of print, use the following calls to SETMODE and WRITE:

```
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 (which was the 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.

The model 5508 line printer has an electronically programmable form length and vertical tabulation stops.

The number of lines in the form is specified by SETMODE function 25 as an integer within the range of 0:126. The default for this setting is 66 lines.

A vertical tab is set with SETMODE function 26 by specifying the line where the tab stop is to occur. By default, a vertical tab is set for each line from 1 through 59 (causing single spacing to be the default case for vertical tabulation). All vertical tabs, with the exception of the tab at line 1, are cleared when you specify a line value of -1 with SETMODE function 26. The vertical tab at line 1 cannot be cleared.

To change the vertical tab stops, the following procedure must be followed exactly:

1. Set the form length (SETMODE function 25).

2. Clear all vertical tabs (SETMODE function 26, <parameter-1> = -1).

3. Set each desired vertical tab using a separate call to SETMODE (function 26, <parameter-1> = (line number minus 1) where the vertical tab is to be set).

4. Issue a top of form (CONTROL operation 1, <parameter> = 0). This forces all tab stops to be set after the form feed is executed. If a subsequent power failure occurs, the tab stops are restored automatically by the system.

5. If necessary, perform operator intervention to align the form.

PROGRAMMING CONSIDERATIONS FOR THE MODEL 5520 PRINTER

The subtype for the model 5520 line printer is 4.

Programmatic Differences Between the Model 5520 and Model 5508

The GUARDIAN software external interface to the model 5520 serial printer is not fully compatible with the interface to the model 5508 serial printer. The major difference between these two printers is the way in which forms control is handled.

The model 5508 printer provides programmable forms length and vertical tab stops, whereas the model 5520 printer provides a 12-channel 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 that 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 and 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.

Using the DAVFU

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.

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).

You can request that the printer skip to the next stop of a particular channel by calling CONTROL operation 1, <parameter> = channel number. The channel number specified must be in the range 0-11; for example:

```
! Skip to channel 0 (top-of-form)
CALL CONTROL (FILENUM, 1, 0);

! Skip to channel 5 (next one-half page)
CALL CONTROL (FILENUM, 1, 5);
```

The 5520 powers on with default DAVFU values. An application program may programmatically change the contents of the DAVFU by calling CONTROLBUF.

The I/O software reinitializes the DAVFU with default values if a CONTROL 1, <parameter> = 0 (skip to top of form) is issued immediately after the file is opened or immediately after a call to SETMODE 28 (reset to configured values). In this manner, the I/O software ensures that the printer is at top of form before loading the default DAVFU. (Loading the DAVFU causes the top of form to be reset to the current line.)

If the previous forms length is different from the default forms length, then the approach just described is not sufficient since the requested top of form will advance the paper the wrong number of lines. For this reason, when loading the printer with standard 66-line (11-inch) paper after nonstandard size paper has been used, it is recommended that the printer be powered off and back on to force the forms length to the default setting.

When using nonstandard size paper, it is recommended that the program require manual intervention to align the paper properly after the DAVFU is loaded.

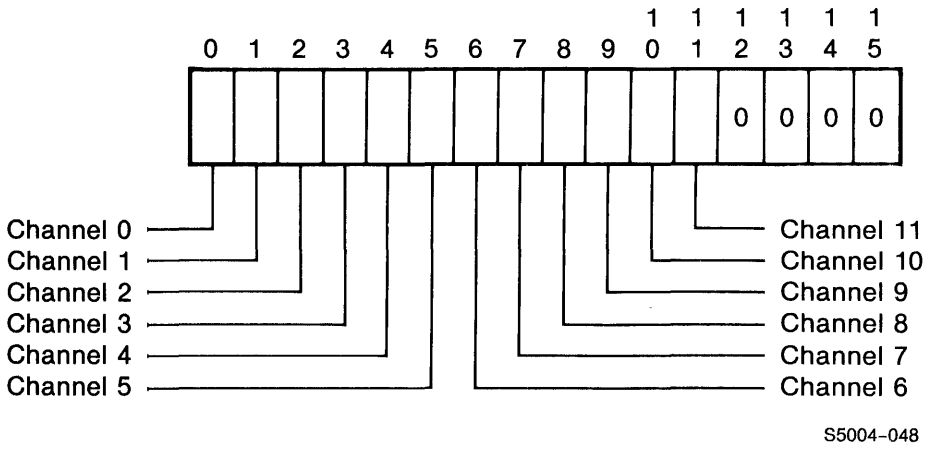
Loading the DAVFU

You can programmatically load the DAVFU by calling CONTROLBUF operation 1. Any previous lines of text will be printed first, before the DAVFU is loaded.

Loading the DAVFU causes the top of form to be reset to the current line.

The <buffer> passed to CONTROLBUF should contain one word for each line on the page. Bits 0 through 11 of each word represent channels 0 through 11, respectively, for that line. A "1" in any of these bits indicates a channel stop in the corresponding channel. Bits 12 through 15 are unused and must be zeros.

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.

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.

Use channel 0 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. Use channel 2 for line termination when spacing and perforation skip are enabled (default). Use channel 11 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.

If single spacing and automatic perforation skip are desired, the DAVFU should contain a channel stop in channel 2 for lines 1 through 60. The I/O software does not use the DAVFU when single spacing without automatic perforation skip.

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.

The recommended procedure for using nonstandard size paper is as follows:

1. Call CONTROLBUF to load the DAVFU.

2. Request manual intervention to load the printer with the desired paper and align it properly.

3. Start writing to the printer.

When reloading the printer with standard 66-line paper after nonstandard size paper has been loaded, the printer should be powered off and back on to reset the forms length to the default setting.

The following example illustrates program code to load the DAVFU. The channel stops specified in the example are the same as those in the 5520 default DAVFU.

```
INT .VFUBUF[0:65] :=
    [%137740, %020000, %030000, %024000, %030000, %020000,
     %034000, %020000, %030000, %024000, %030400, %020000,
     %034000, %020000, %030000, %025000, %030000, %020000,
     %034000, %020000, %030400, %024000, %030000, %020000,
     %034000, %020000, %030000, %024000, %030000, %020000,
     %037400, %020000, %030000, %024000, %030000, %020000,
     %034000, %020000, %030000, %024000, %030400, %020000,
     %034000, %020000, %030000, %025000, %030000, %020000,
     %034000, %020000, %030400, %024000, %030000, %020000,
     %034000, %020000, %030000, %024000, %030000, %060000,
     %000000, %000000, %000020, %000000, %000000, %000000];
        .
        .

CALL CONTROLBUF (FILENUM, 1, VFUBUF, 132);
```

Underline Capability

The model 5520 printer has the ability to automatically do a partial line feed before underlining. This feature is desirable so that the underscore characters do not overlap the bottom parts of the characters being underlined. A partial line feed is performed following a carriage return (%15) only if the next line contains only underscore (%137) and space (%40) characters.

You can create an application program to use this underlining capability, accomplished by overprinting, as follows:

1. Turn off spacing using SETMODE function 6, <parameter-1> = 0.

2. Write the desired text.

3. Turn spacing back on by calling SETMODE function 6, <parameter-1> = 1.

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 is selected by calling SETMODE function 68, <parameter-1> = 1.

The expanded print option (double width) provides a horizontal pitch of 5 characters per inch. Your application program can select expanded print by calling SETMODE function 68, <parameter-1> = 2.

Reenable normal printing by calling SETMODE function 68, <parameter-1> = 0; for example:

```
! Select condensed print
CALL SETMODE (FILENUM, 68, 1);

! Write the text
CALL WRITE (FILENUM, DATABUFFER, COUNT);

! Select expanded print
CALL SETMODE (FILENUM, 68, 2);

! Write the text
CALL WRITE (FILENUM, DATABUFFER, COUNT);

! Go back to normal print
CALL SETMODE (FILENUM, 68, 0);
```

The I/O software adjusts the maximum buffer size according to whether expanded, condensed, or normal print is enabled. This maximum is computed as follows:

- Normal Print Mode--the configured record size
- Expanded Print Mode--one-half the configured record size
- Condensed Print Mode--1.67 times the configured record size (rounded down, maximum 218 bytes)

For example, if you configure the device with a record size of 132 bytes, the maximum buffer size allowed in condensed print mode is 218 bytes. The same configuration allows only 66 bytes for writes to the printer in expanded print mode. Furthermore, if the device is configured with a record size of 80 bytes, the maximum buffer size allowed in condensed print mode is 133 bytes. The same configuration allows only 40 bytes for writes to the printer in expanded print mode.

Error Conditions for the Model 5520

The following paragraphs summarize and explain the meaning of some of the file-system errors when they occur with a model 5520 printer.

Error 21 (ILLEGAL COUNT) occurs when the byte count for a write is greater than the configured record size for normal print mode, one-half the configured record size for expanded print mode, or 1.67 times the configured record size (rounded down, maximum 218 bytes) for condensed print mode. This error also occurs when the byte count specified for a DAVFU load is not an even number or is greater than 508 bytes (254 lines).

Error 100 (DEVICE NOT READY) generally means that the device is offline, and the ON LINE button must be pressed.

Error 102 (PAPER OUT) on this printer means either the printer is out of paper or the bail has not been closed; the operator must correct the problem.

Error 104 (NO RESPONSE FROM DEVICE) means the printer did not return the requested status; either the printer power is off, or there is a hardware problem (for example, a parity error occurred on each request for status until the retry count was exhausted).

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 nonrecoverable 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) occurs for one of the following reasons:

1. Invalid status returned from printer
2. "Buffer full" status lasted too long
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. See "DEVICE POWER ON Error" below.

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 through 37 octal (%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: (%12) line feed, (%14) form feed, (%15) carriage return, and (%33) escape. 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 nonstandard character (%206), followed by paper movement equivalent to one line feed.

DEVICE POWER ON Error

If the printer power fails and then powers up again, the previous contents of the DAVFU and all data in the printer's internal buffers (approximately 1K bytes) are lost. The printer powers on in an offline state, and the DAVFU is reloaded with default values.

File-system error 100 (DEVICE NOT READY) will first be returned to the application. If the power on occurred while the file was open, error 191 (DEVICE POWER ON) is returned after the printer has been placed online.

If recovery is to be attempted from a POWER ON condition, CONTROLBUF should be called, if necessary, to reload the DAVFU with user values. Manual intervention should then be requested in order to align the paper properly and to visually guarantee a good restart point.

If the printer is powered on without paper loaded, error 100 (DEVICE NOT READY) is first returned. If the ON LINE button is pressed at this time without paper loaded, then error 102 (PAPER OUT) is returned. Once the Paper Out condition is cleared and the ALARM/CLEAR button is pressed, the printer again becomes Not Ready. The ON LINE button must be pressed. At this point, error 191 (DEVICE POWER ON) is returned if the power on occurred while the file was open.

PROGRAMMING CONSIDERATIONS FOR THE MODEL 5530 PRINTER

The subtype for the model 5530 line printer is 6.

The 5530 printer is available only on NonStop systems.

The 5530 printer powers on in full status mode; partial status mode is not supported on the 5530, and status is returned to the host only when requested.

The 5530 printer does not support a format control tape or a programmable VFU (vertical format unit); forms control operations are simulated by the I/O software. CONTROL operation 1 allows forms control and line skipping.

The baud rate for use with the 5530 printer can be set by SETMODE function 22 to one of the following: 75, 150, 300, 600, 1200, 2400, 4800, or 9600. (No other line speeds are supported for the 5530.) Default is the SYSGEN-configured baud rate or 9600 baud if not specified on the SYSGEN.

A cut sheet paper feeder option is available for the 5530.

USING A MODEL 5508, 5520, OR 5530 PRINTER OVER A TELEPHONE LINE

A model 5508, 5520, or 5530 line printer can be used over a phone line using a Bell System 103, 113, or 212 modem at 300 or 1200 BAUD. Two answering modes are available: AUTOANSWER (the default) and CTRLANSWER.

In AUTOANSWER mode, DATA TERMINAL READY is raised by the I/O software, allowing incoming calls from the printer to be answered at any time. Once the telephone has been answered, an application can write to the printer without any special modem control code, provided the connection is still good. This allows any program that can access the printer locally to access it remotely over a modem as well. For example:

```
:FILES /OUT $LPMODEM/
:FUP COPY <filename>, $LPMODEM
```

In AUTOANSWER mode, the communication line is not disconnected when the file is closed.

CTRLANSWER mode gives the user full control of answering and hanging up the phone. DATA TERMINAL READY is not raised until the user process does a CONTROL operation 11 (answer the phone), which waits for the modem to connect with the printer. When the file is closed or the user process does a CONTROL operation 12 (hang up the phone), DATA TERMINAL READY is dropped and the line is hung up; for example:

```
CALL CONTROL (FILENUM, 11);    ! answer the telephone
        .
        .
```

```
CALL CONTROL (FILENUM, 12);   ! hang up the telephone
```

AUTOANSWER or CTRLANSWER mode can be specified as configuration parameters in SYSGEN (see the System Management Manual) or by using SETMODE function 29:

```
CALL SETMODE (FILENUM, 29, 0); ! CTRLANSWER mode
        .
        .
CALL SETMODE (FILENUM, 29, 1); ! AUTOANSWER mode
```

Unlike the 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, 5520, or 5530 printer:

LP55nnM	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 System Management Manual.

ERROR RECOVERY

The following errors require special consideration for all line printers:

100	device not ready
200-255	path errors

Consideration is also necessary when using nowait 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.

Not Ready

It is the responsibility of the application process to handle "not ready" and/or "paper out" conditions. With some printers, either condition causes a "not ready" indication (see the operating manual for the specific printer). Typically, if either of these conditions occurs, a message indicating the condition should be displayed on the home terminal (the logical device

number of the home terminal can be obtained using the MYTERM process control procedure). See the System Procedure Calls Reference Manual for the syntax of the MYTERM procedure. Your application process should then wait for the terminal operator to respond, indicating that the printer is ready. For example:

```
  .
LITERAL NOT^READY = 100,
        PAPER^OUT = 102;
  .
RETRY:
CALL WRITE ( PRINTER, BUFFER, 132 );
IF < THEN ! error occurred.
  BEGIN
    CALL FILEINFO ( PRINTER, ERROR );
    IF ERROR = NOT^READY OR ERROR = PAPER^OUT THEN
      BEGIN
        BUFFER ':=' "** PRINTER NOT READY";
        CALL WRITEREAD ( HOME^TERM, BUFFER, 20, 1, NUM^READ );
          .
          The application program informs the terminal
          operator then waits for a reply
          .
        GOTO RETRY;
      END
    ELSE .....; ! trouble.
  END;
```

Path Errors

Path error recovery on a printer requires some special considerations because of paper movement.

If a path error is detected and is either error 200 or 201, the operation never got started. These errors can simply be retried.

If a path error is detected and is one of errors 210 through 231, the operation failed at some indeterminate point, and paper movement may have occurred. Depending on the application, different approaches to error recovery are necessary. If the operation is a critical one, such as the printing of payroll checks, the check should probably be cancelled and a message sent to the operator. However, if the information being printed is not considered critical, the line can be reprinted (and may thus be duplicated).

SUMMARY OF PRINTER CONTROL, CONTROLBUF, AND SETMODE OPERATIONS

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 2	(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

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

     <parameter> = none

12 = line printer (subtype 3 or 4), disconnect the modem:

     <parameter> = none
```

Line Printer CONTROLBUF Operations

```
 1 = line printer (subtype 4), load DAVFU:

     <buffer> = VFU buffer to be loaded

     <count>  = number of bytes contained in <buffer>
```

Line Printer SETMODE Operations

```
<function>

 5 = set system automatic perforation skip mode (assumes
     standard VFU function in channel 2):

     <parameter-1> = 0, off - 66 lines per page
                   = 1, on  - 60 lines per page (default)

     <parameter-2> is not used.

 6 = set system spacing control:

     <parameter-1> = 0, no space
                   = 1, single space (default setting)

     <parameter-2> is not used.
```

```
<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
                    16, baud rate = 200

     <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 or clear vertical tabs:

     <parameter-1> >= 0, (line#-1) of where tab is to be set
                   = -1, clear all tabs (except line 1)

        Note:  A vertical tab stop always exists at line 1
               (top of form).

     <parameter-2> is not used.

27 = set system spacing mode:

     <parameter-1>.<15> = 0, postspace (default setting)
                        = 1, prespace

     <parameter-2> is not used.
```

```
<function>

28 = reset to configured values:

     <parameter-1> = 0

     <parameter-2> is not used.

29 = line printer (subtype 3 or 4), set automatic answer
     mode or control answer mode:

     <parameter-1>.<15> = 0, CTRLANSWER
                        = 1, AUTOANSWER (default)

     <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          } 0 = not moving
        ** RESERVED FOR LATER USE **     } 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
```

```
<function>

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    } 0 = partial status
                status  } 1 = full status
                field   } 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
```

<function>

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

<last-params>[0] for printer (subtype 4) (cont'd)

.<15> = device not ready } 0 = ready
} 1 = not ready

All other bits are undefined.

<last-params>[1] = auxiliary status word if
<last-params>[0].<9:11> = 7; otherwise 0.
Auxiliary status word is as follows:

.<9:13> = auxiliary status } 0 = no errors this field
} 1 = no shuttle motion
} 2 = character generator absent
} 3 = VFU channel error
} 4-31 = reserved for future use

.<14:15> } always 3

All other bits are undefined.

68 = line printer (subtype 4), set horizontal pitch:

<parameter-1> = 0, normal print (default)
= 1, condensed print
= 2, expanded print

<parameter-2> is not used.

NOTE

SETMODE 29 (Set Auto Answer or Control Answer Mode) is the only SETMODE function not affected by a SETMODE 28 or a RESET on OPEN.

SECTION 8

INTERFACING TO MAGNETIC TAPES

This section describes the characteristics of magnetic tape as a storage medium for files and the procedures used to control tape usage. Concepts, programming considerations, and CONTROL operations for use with magnetic tape drives are described. BCD/ASCII character set equivalents are shown; available conversion modes are also illustrated.

The file system provides for data transfers between magnetic tape files and application processes in records of 24 to 32767 bytes.

GENERAL CHARACTERISTICS OF MAGNETIC TAPE FILES

Individual files on magnetic tape are not accessed explicitly; instead, the magnetic tape unit itself is accessed either 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 (EOF) (note that closing a magnetic tape file following a write to tape does not write an EOF mark).

The CONTROL operations and the CLOSE procedures each provide four options for rewinding tape. The CONTROL options are explained in Table 8-1. The CLOSE procedure is explained in the System Procedure Calls Reference Manual.

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 (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 systems other than Tandem to be read or edited). WRITEUPDATEs are not allowed on the 5106 Tri-Density Tape Drive and on the 5107 Tape Drive. To ensure the integrity of data read from tape, however, the minimum read operation should be for at least 24 bytes.

Multireel files, if desired, must be implemented by the application program.

The device type for a magnetic tape unit is 4.

All programs supplied with the release, when accessing a magnetic tape unit, open it with exclusive access such as (OPEN, <flags>.<9:11> = 1).

The maximum number of bytes permitted in a single-file system transfer to a tape file depends on the system type and the model of the controller to which the tape is connected:

Controller Model	Maximum Transfer Length
P/N 3201	32767 bytes
P/N 3202	32767 bytes
P/N 3206	32767 bytes
P/N 3207	32767 bytes

NOTE

An anomaly exists when using the model 3202 controller. If a read of 4096 bytes is requested and the tape contains exactly 4095 bytes, a <read-count> of 4096 is returned. In all other cases, the <read-count> is correct.

The recommended maximum tape record block sizes using the BACKUP program are as follows:

Density	(bpi)	Block Size
NRZI	800	4096 bytes
PE	1600	8192 bytes
GCR	6250	30720 bytes

Maximum record block size on NonStop systems actually is 32767 bytes for application programs but 30720 bytes for disc BACKUP and RESTORE.

The maximum recommended tape record sizes for application programs are as follows:

Density (bpi)		Record Size
NRZI	800	4096 bytes
PE	1600	8192 bytes
GCR	6250	32767 bytes

NOTE

The default record length specified in SYSGEN is totally ignored by the I/O software for magnetic tapes.

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 a 5106 or 5107 tape drive).

CONTROL — executes the following operations to a magnetic tape:

- write end-of-file mark
- rewind (load or unload, online or offline, wait or 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 and clears the translation technique option (7-track tape only) and the short-write treatment option. Selects tape density for 5106 Tri-Density Tape Drive.
SETMODENOWAIT	is used the same as SETMODE except in a nowait 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:

```
    .
INT .TAPE^NAME[0:11] := ["$TAPE1", 9 * ["  "]];
     TAPE^FNUM,
    .TAPE^BUF [0:1023],
     NUM^WRITTEN,
     NUM^READ;
  .
  .
CALL OPEN ( TAPE^NAME, TAPE^FNUM, . );
IF < THEN ... ; ! error occurred.
  .
```

Then, to write a record on tape:

```
  .
CALL WRITE ( TAPE^FNUM, TAPE^BUF, 2048, NUM^WRITTEN );
  .
```

2048 bytes of TAPE^BUF are written on tape, the value 2048 is returned in NUM^WRITTEN

To terminate access to a magnetic tape unit, call the CLOSE procedure. Closing the tape also causes the tape to be rewound and the tape unit to be placed offline. Options to CLOSE permit the magnetic tape unit to be left online, for no rewind to occur, for tape to be unloaded (and the unit taken offline), and wait or don't wait for completion.

For example, to CLOSE a magnetic tape unit and cause tape to be rewound and unloaded, the following is written in an application program:

```
  .
LITERAL REW^UNLOAD = 0;

  .
CALL CLOSE ( TAPE^FNUM, REW^UNLOAD );
  .
```

The rewind and unload operation is initiated. CLOSE returns immediately to the application program and the rewind operation is performed concurrently with application program execution. (If the magnetic tape is again opened, any read, write, or control operation results in an error indication. A subsequent call to FILEINFO returns error 100 (device not ready)).

MAGNETIC TAPE CONCEPTS

Associated with operations involving magnetic tapes are:

- Beginning-of-Tape (BOT) and End-of-Tape (EOT) markers
- Files
- Records

BOT and EOT Markers

The BOT and EOT 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 EOF mark. The programmer must ensure that the application program explicitly writes an EOF mark on tape (using CONTROL, operation 2) to terminate a file.

The application program must also 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 (identified by EOF marks).

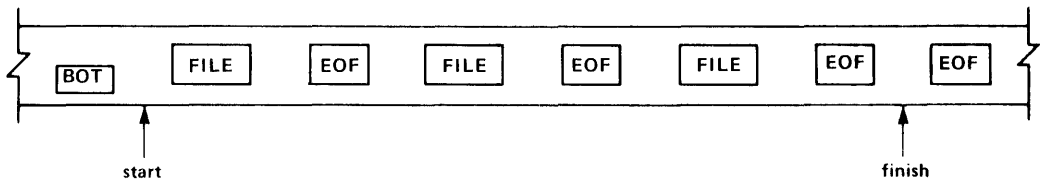
There are two considerations when spacing files:

- 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.

The following examples show how tape is positioned in relation to the read-write heads after performing various file spacing operations.

Example 1 illustrates the space forward (CONTROL operation 7) of three files:

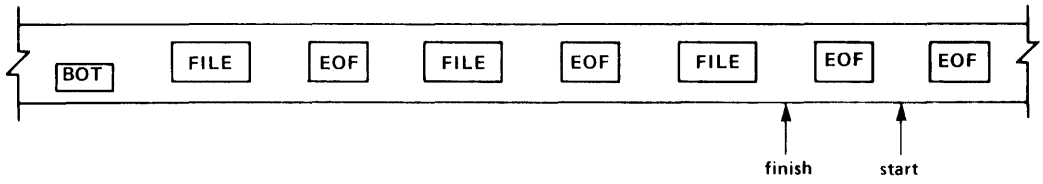
```
CALL CONTROL ( TAPE^FILE, 7, 3 );
```



S5004-049

Example 2 illustrates a space backward of one file from the finish point in the preceding example:

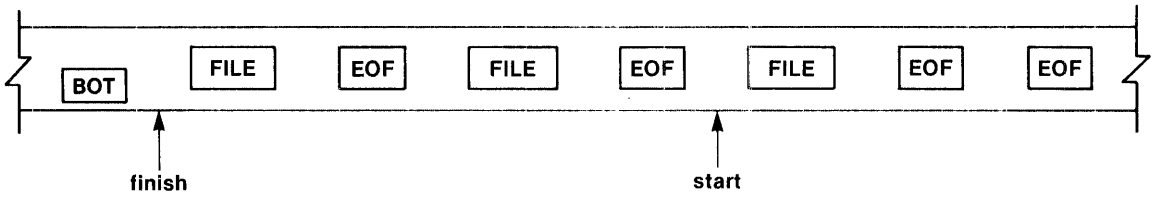
```
CALL CONTROL ( TAPE^FILE, 8, 1 );
```



S5004-050

Example 3 illustrates a space backward of 10 files from the finish point in the preceding example:

```
CALL CONTROL ( TAPE^FILE, 8, 10 );
```



S5004-051

Backward spacing of files stops at BOT.

Records

Data is stored on magnetic tape in records containing from 24 to 32767 data bytes (plus associated hardware-generated check information). The file system does not provide access to individual elements within a record.

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 operations are not allowed on the 5106 or 5107.

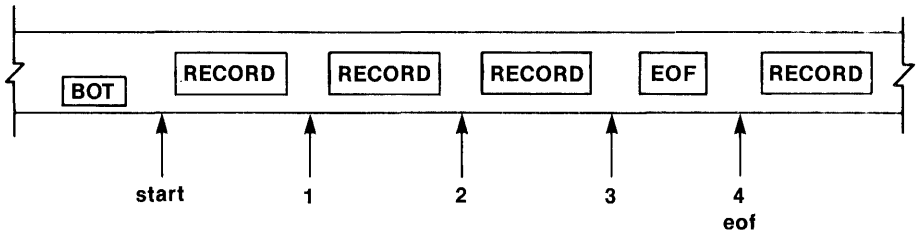
It is important to note that the new record written by the WRITEUPDATE procedure must be exactly the same size as the record being replaced; otherwise, a subsequent error will occur. Also, there is a practical limit of five on the number of times WRITEUPDATE should be performed on the same record.

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.

Consider the following 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;
```

The first, second, and third READs 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.



S5004-052

If the value passed in the <read-count> parameter is not sufficient to read an entire record, an error indication is returned to the application. For example, a record on tape contains 1024 data bytes and a read of 256 bytes is executed:

```
    .
CALL READ ( TAPE^NUM, BUFFER, 256, NUM^READ );
IF < THEN ...; ! error occurred.
    .
```

256 bytes are transferred into BUFFER, 256 is returned to NUM^READ, and the condition code indicator is set to CCL. A subsequent call to FILEINFO would return ERROR = 21 (illegal count specified). After the read, the tape is positioned with the read head preceding the next record on tape.

Individual records on tape can be edited by using the READ and WRITEUPDATE procedures. (WRITEUPDATE is not allowed on the 5106 or 5107 tape drives.) For example, a record to be edited is read from tape:

```
.
CALL READ ( TAPE^FNUM, TAPE^BUF, 2048, NUM^READ );
.
```

NUM^READ characters are read from tape and transferred into TAPE^BUF.

The application makes the necessary changes to the record in TAPE^BUF, then edits the tape by calling the WRITEUPDATE procedure (except on the 5106 and 5107).

```
.
CALL WRITEUPDATE (TAPE^FNUM, TAPE^BUF, NUM^READ, NUM^WRITTEN);
.
```

The tape is backspaced over the record just read, then updated by writing the new record in its place. NUM^READ, from the preceding read, specifies the number of bytes to be written (ensuring that exactly the same number is written that were previously read).

The file system provides (as a parameter to the CONTROL procedure) the ability to space forward and backward a specified number of records.

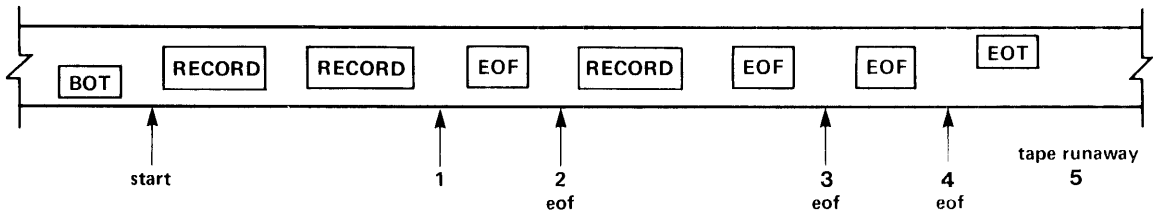
There are a number of considerations when spacing records:

- 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 illustrates repeated space forwards (CONTROL operation 9) of two records (tape is positioned at BOT):

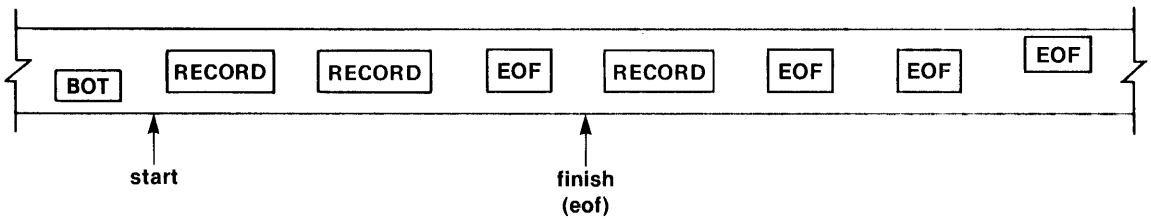
```
loop:  CALL CONTROL ( TAPE^FILE, 9, 2 );
       GOTO loop;
```



S5004-053

Example 2 illustrates a space forward of 10 records (tape is positioned at BOT):

```
CALL CONTROL ( TAPE^FILE, 9, 10 );
```

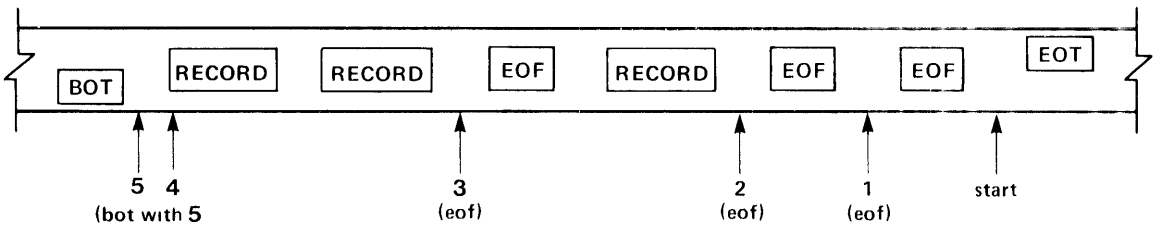


S5004-054

The operation stops when an EOF mark is read.

Example 3 illustrates repeated backward space records (CONTROL operation 10) (tape is positioned past the last EOF mark):

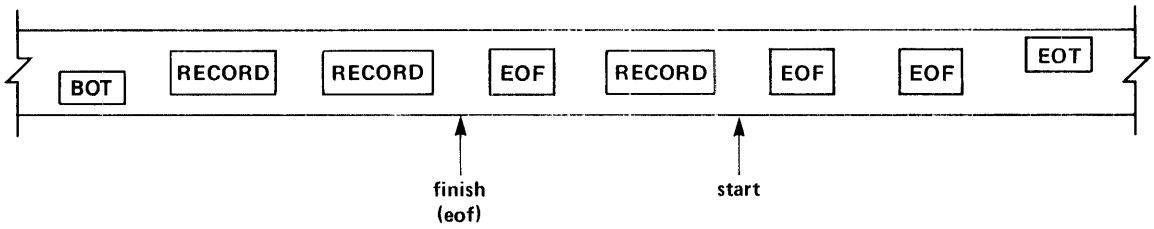
```
LOOP:  CALL CONTROL ( TAPE^FILE, 10, 2 );
       GOTO LOOP;
```



S5004-055

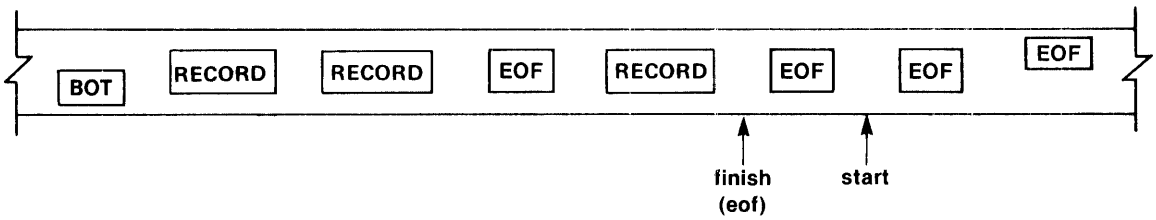
Example 4 illustrates a backward space records of 10 records (tape is positioned as shown):

```
CALL CONTROL ( TAPE^FILE, 10, 10 );
```



S5004-056

Example 5 illustrates the same operation as the preceding example (space backward 10 records) but with the tape positioned just past an EOF mark:



S5004-057

The backspace stops when an EOF mark is encountered.

PROGRAMMING CONSIDERATIONS FOR THE 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 the recording density may 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. In other words, 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 that 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 down-loaded microcode. An explicit request to download the controller microcode can be made by the operator using the PUP LOADMICROCODE command. Refer to your System Operations Manual.

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 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 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 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 RECOVERY

Most error recovery is attempted automatically by the file system. However, because of tape movement, there are special considerations when an error occurs with a magnetic tape unit.

If a tape unit's power fails, file-system error 100 (not ready), error 212 (EIO instruction failure) or error 218 (interrupt timeout) is returned to the application program when any read, write, or control operation is attempted. After power is restored and the tape unit is again accessed, a subsequent call to FILEINFO returns error 153 (tape drive power on). It is the application's responsibility to ensure that the correct tape is loaded following a power failure.

Tape units, if a tape is loaded, are automatically put back into an operating (ready) state when power is restored. (See error 153 below for the 5106 Tape Subsystem.) On the 5103 and 5104 tape drives, if two units are set to the same unit number when power is restored, an arbitrary decision is made as to which one is subsequently placed online. The implication is that it may not be the desired unit. Therefore, it is recommended that when an application program finishes with a tape unit, a rewind and unload operation be performed. Then, only previously active units return to a ready state.

The following list includes the most commonly encountered tape errors. A complete list of file-system errors appears in the System Messages Manual.

- Error 100 (Device Not Ready)

This error indicates that the device is not online. Error 100 is always returned if the drive is accessed while rewinding; wait until the rewind completes. This error may also indicate a possible tape drive, formatter, or controller power failure. (See error 153.)

- Error 120 (Nonrecoverable Data Parity Error)

For the 5106 Tape Subsystem, if the controller detects a data error during a read or write operation, it automatically retries the operation--up to 50 times for read requests and up to 40 times for write requests. If the retry operation succeeds, the application program does not receive an error indication. If, however, the retry does not succeed, the tape is positioned after the invalid record, an error message appears at the operator console, and file-system error 120 (data parity error) is

returned to the application program. For the other tape drives, reads and writes are retried 40 times.

- 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 and 5107 tape drives. If the operating system cannot locate either of the controller microcode files, cannot read either file 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.)

- Error 212 (EIO Instruction Failure)

A controller 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 and 5107 tape drives. Certain errors cause the controller to respond with the same error indication until the controller is reset by a PUP LOADMICROCODE command, by processor reset, or by power failure, returning file-system error 224 (controller error) to the application program. See "Controller Self-Test Failure" immediately preceding this discussion.

Path Errors

If a path error is detected and is either error 200 or 201, the operation never got started (the tape did not move).

If a path error is detected and is one of the errors 210 through 231, the operation failed at some indeterminate point. Therefore, tape motion may have occurred. There are a number of ways to handle this type of error:

- If a path error occurs while writing, backspace and reread the last record. If a parity error occurs, backspace again and rewrite the last record.

- Keep track of the number of records read or written on tape. Then if an error of this type occurs, rewind the tape and space forward the appropriate number of records, and reinitiate the operation.

- If writing, write a sequence number as part of each record written. If one of these errors occurs, retry the operation and continue. Then when reading the tape, discard all but the last record containing duplicate sequence numbers.

- If reading, and sequence numbers were written on tape, keep track of the sequence number of the current record. Then if a path error occurs, retry the operation. If the expected sequence number is not read, meaning that a record was skipped over when the path error occurred, backspace the tape two records.

Additional considerations are necessary if nowait I/O is used. Refer to the appropriate procedure description in the System Procedure Calls Reference Manual for details.

SUMMARY OF MAGNETIC TAPE CONTROL OPERATIONS

Table 8-1. 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}

SEVEN-TRACK MAGNETIC TAPE CONVERSION MODES

The 7-track tape drive supports four data-conversion modes:

- ASCIIBCD
- BINARY3TO4
- BINARY2TO3
- BINARY1TO1

These conversion modes determine how the data is translated while writing to or reading from tape. Any one of the four may be selected in the system configuration or using the SETMODE procedure.

ASCIIBCD translates between 8-bit memory ASCII characters and 6-bit tape BCD characters. The translation is the same for both uppercase and lowercase letters. Table 8-2 illustrates the BCD character set and the equivalent ASCII characters. Any ASCII characters other than those shown in table 8-2 are translated to a BCD space character, which is %20 on tape. Although the Tandem system does not support BCD memory characters, the character set is included in table 8-2 as a reference.

Table 8-2. ASCII Equivalents to BCD Character Set
(Continued on next page)

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

Table 8-2. ASCII Equivalents to BCD Character Set (continued)

BCD TAPE (OCTAL)	BCD MEMORY (OCTAL)	CHARACTER	ASCII (OCTAL)
45	45	N	116
46	46	O	117
47	47	P	120
50	50	Q	121
51	51	R	122
52	52	]	135
53	53	$	44
54	54	*	52
55	55	)	51
56	56	;	73
57	57	^	136
60	20	&	46
61	21	A	101
62	22	B	102
63	23	C	103
64	24	D	104
65	25	E	105
66	26	F	106
67	27	G	107
70	30	H	110
71	31	I	111
72	32	[	133
73	33	.	56
74	34	<	74
75	35	(	50
76	36	+	53
77	37	!	41

The maximum record size in the ASCIIBCD conversion mode is 32767 bytes; the parity is even.

BINARY3TO4

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 24576 bytes; parity is odd. Use the number of 8-bit memory bytes to specify a byte count in a read or write.

BINARY2TO3

BINARY2TO3 converts each block of two 8-bit bytes to three 6-bit tape characters. This conversion mode results in two bits being unused on tape for every three tape characters.

The following example illustrates the use of this conversion mode.

A0:A7 and B0:B7 represent two 8-bit memory bytes.

```
A0 A1 A2 A3 A4 A5 A6 A7
B0 B1 B2 B3 B4 B5 B6 B7
```

When writing to tape, these two bytes become

```
00 00 A0 A1 A2 A3
A4 A5 A6 A7 B0 B1
B2 B3 B4 B5 B6 B7
```

When reading from tape, these bytes

```
00 00 A0 A1 A2 A3
A4 A5 A6 A7 B0 B1
B2 B3 B4 B5 B6 B7
```

become

```
A0 A1 A2 A3 A4 A5 A6 A7
B0 B1 B2 B3 B4 B5 B6 B7
```

The maximum record size in the BINARY2TO3 conversion mode is 21845 bytes; the parity is odd. Use the number of 8-bit memory bytes to specify a byte count in a read or write.

BINARY1TO1

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 32767 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 the System Procedure Calls Reference Manual for a 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 your System Management Manual.

SELECTING SHORT WRITE MODE

Function 52 of the SETMODE procedure selects the short write mode. The values that specify the short write modes are:

<parameter-1> = 0, allows writes shorter than 24 bytes (default).

1, does not allow writes shorter than 24 bytes.

<parameter-2> is not used.

NOTE

When short writes are disallowed, an attempt to WRITE or WRITEUPDATE a record that is shorter than 24 bytes causes error 21 (bad count) to be issued.

See the System Procedure Calls Reference Manual for a discussion of the SETMODE procedure.

SECTION 9

INTERFACING TO CARD READERS

Card readers are not often used in transaction processing. The interface to card reader capability is available to retain compatibility with other, older systems. A general overview of card reader usage is presented here to provide an understanding of the process.

The file system provides for transfers of data from card readers to application processes in blocks of 0 characters (skip card) to the maximum number of characters required to read a card.

GENERAL CHARACTERISTICS OF CARD READERS

Card readers can be accessed either by $<device-name> or $<logical-device-number>.

There are three read modes: ASCII, column-binary, and packed-binary.

End-of-file indication is available only 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 software 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 nowait manner on an open file.
READ	is used to read a card.
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.
CLOSE	stops access to an open file.

READ MODES

Data can be transferred from the card reader to the application process in one of three read modes: ASCII, column-binary, or packed-binary.

ASCII is the default mode when a card reader is first opened. In ASCII mode, the card is assumed to have been encoded using Hollerith code. Each column is converted into its ASCII equivalent (one byte per column). The conversion conforms to the "Hollerith Punched Card Code" described in the ANSI document ANSI X3.26-1970.

If the Hollerith equivalent to EOF! followed by 76 blanks is read, an end-of-file indication is returned to the application process. This indication is returned regardless of the number of bytes requested in the call to READ. Note that the "!" character of the end-of-file indication can be represented by either the Hollerith code 11,8,2 ("!" on the IBM 029 keyboard) or the Hollerith code 12,8,7 ("|" on the IBM 029 keyboard).

In the ASCII mode, a <read-count> of 80 is required to fully read an 80-column card.

When using an IBM 029 keypunch, you must make a keyboard translation to enter certain ASCII characters. These ASCII characters and their corresponding 029 keys are:

ASCII Character	029 Key	
[(%133)	¢	
\ (%134)	0.8.2	
] (%135)	!	
^ (%136)	¬	(logical NOT)
! (%041)	|	

Column-binary mode is set through use of SETMODE function 21, <parameter 1> = 1. In the column-binary mode, each column image is returned right justified in one word (two adjacent bytes). Word.<0:3> is 0; the top row of the card is returned in word.<4>. In the column-binary mode, a <read-count> of 160 is required to fully read an 80-column card.

Column-binary read mode is illustrated in Figure 9-1.

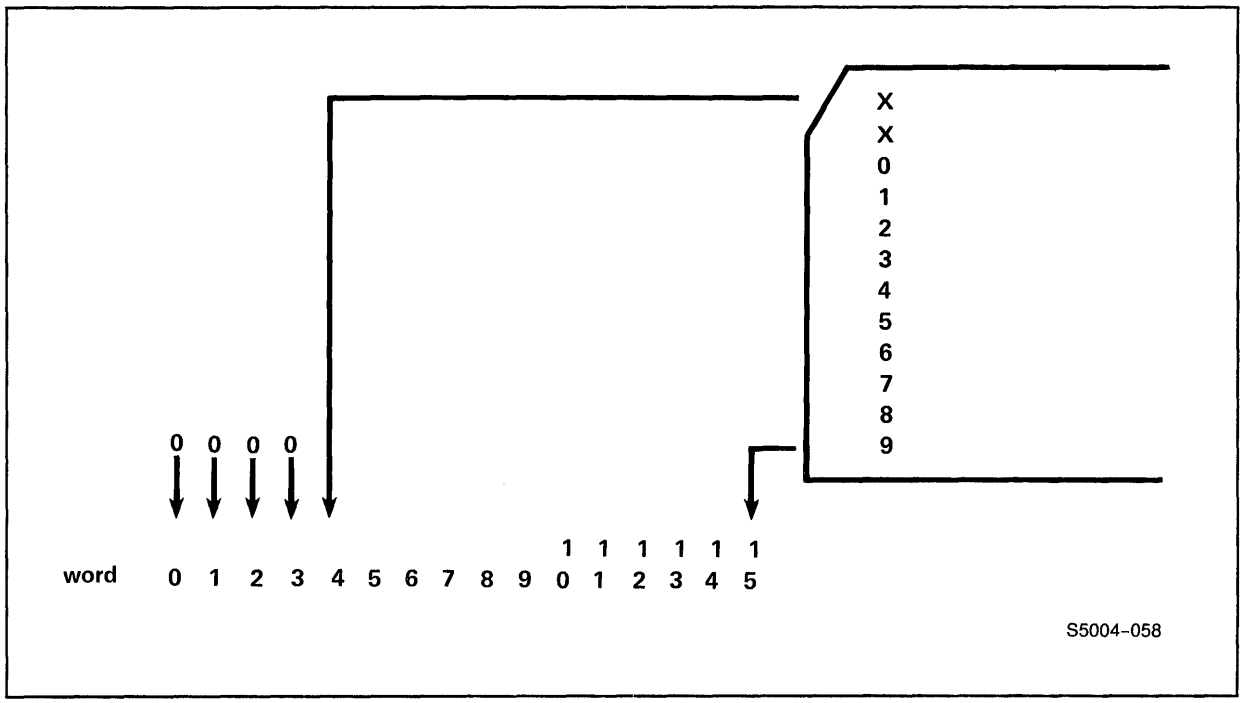


Figure 9-1. Column-Binary Read Mode for Cards

Packed-binary mode is set through use of SETMODE function 21, <parameter 1> = 2. In the packed-binary mode, the card image is returned to the application process's buffer as a contiguous stream of bits (each column returns twelve bits). The top row of the card has the greatest numerical significance.

In the packed-binary mode, a <read-count> of 120 is required to fully read an 80-column card. If the specified <read-count> is such that the last column read does not fill the last word returned to the application process's buffer, the unfilled part of the last returned word is zeroed.

Packed-binary read mode is illustrated in Figure 9-2.

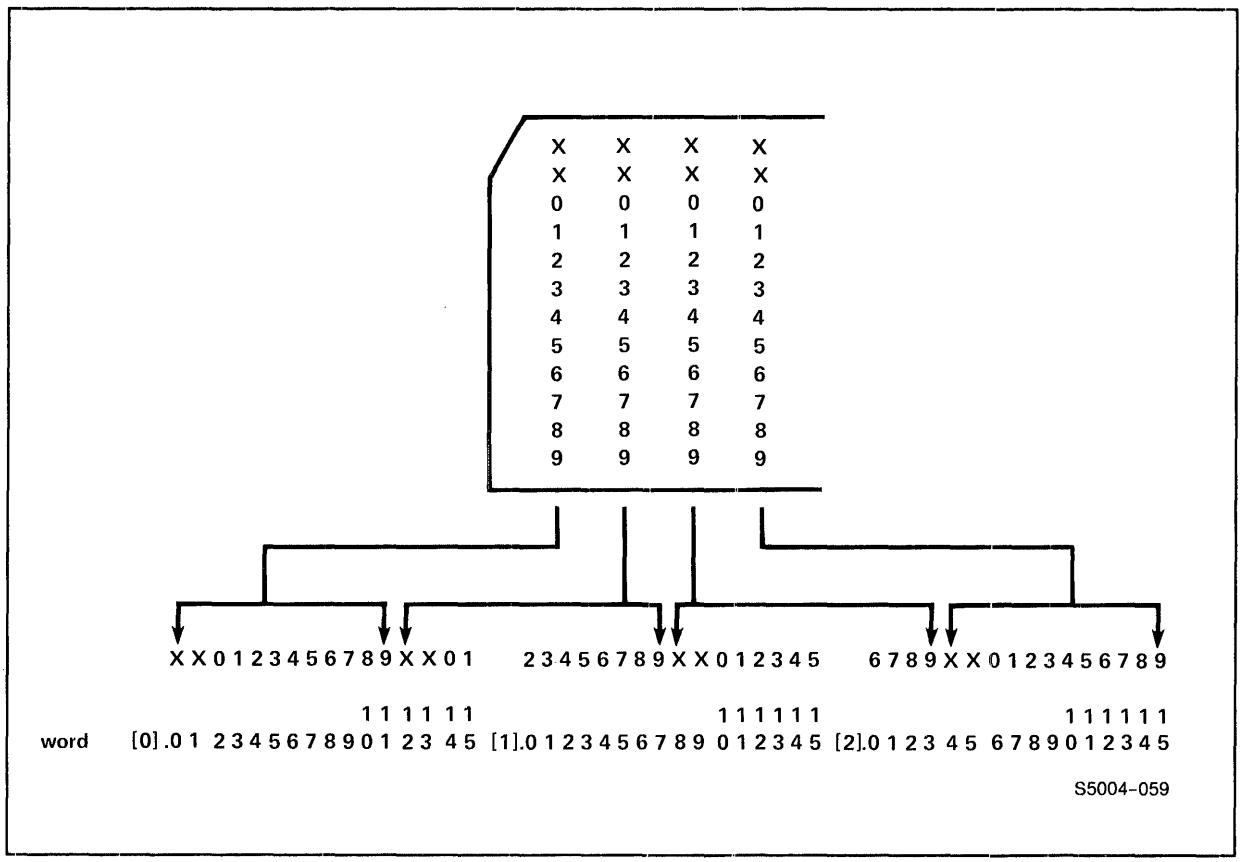


Figure 9-2. Packed-Binary Read Mode for Cards

ACCESSING A CARD READER

First, like any other file, the card reader must be opened:

```
INT .CARD^FNAME[0:11] := ["$CARDRDR",          !data
     8 * ["  "]],                              !declarations.
     CARD^FNUM,
     CARD^BUFFER[0:39];

LITERAL EXCL^ACC = %20;


CALL OPEN ( CARD^FNAME, CARD^FNUM, EXCL^ACC );
IF <> THEN ... ; ! error.
   .
```

Then, to read a card, call the READ procedure:

```
   .
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, make the following call to SETMODE:

```
LITERAL SET^READ^MODE = 21,
        PACKED^BINARY =  1;

CALL SETMODE ( CARD^FNUM, SET^READ^MODE, PACKED^BINARY );
IF < THEN ... ; ! error.
```

further reads of $CARDRDR return the card image in packed-binary format.

To close the card reader, call the CLOSE procedure:

```
   .
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.

It is the responsibility of the application process to handle the NOT READY condition. Typically, if this condition occurs, a message indicating the not ready condition, should be displayed on the home terminal (the device name of the home terminal is obtainable with the MYTERM process control procedure). The application process should then wait for the terminal operator to respond, indicating that the card reader is ready; for example:

```
  .
LITERAL NOT^READY = 100;
  .
RETRY:
CALL READ ( CARD^FNUM, CARD^BUFFER, 80 );
IF > THEN ... ! end-of-file.
ELSE
IF < THEN     ! error occurred.
  BEGIN
    CALL FILEINFO ( CARD^FNUM, ERROR );
    IF ERROR = NOT^READY THEN
      BEGIN
        BUFFER ':=' "** CARD READER NOT READY";
        CALL WRITEREAD ( HOME^TERM, BUFFER, 24, 1, NUM^READ );
          .
```

The application process informs the terminal operator, then waits for a reply.

```
          .
        GOTO RETRY;
      END
    ELSE .....; ! trouble.
  END;
  .
```

Motion Check

The motion check error indicates that the card reader hardware has signalled a motion check. One cause of this condition is a momentary loss of power to the card reader while the card is being transported through the read station.

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, consider the error to be fatal.

Read Check

The read check error 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, 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 through 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, and 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 an error 210 through 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.

SECTION 10

INTERFACING TO THE OPERATOR CONSOLE

The operator console is used to log the occurrence of system error conditions, to log system statistical information, and to log application-supplied information. This information is handled by the operator process, which is always running. The operator process automatically sends log information to all devices enabled to receive console messages. Any process can send messages to the operator console through the use of standard file-system procedures.

There are three places where console messages may be directed:

1. A console terminal device

   For NonStop systems, this may be a hard-copy device or it may be the Operations and Service Processor (OSP). Console message logging can be redirected to other devices, and may be disabled. (See "Using Console Messages" in your System Operations Manual).

2. A disc log file designated $SYSTEM.SYSTEM.OPRLOG

   All messages are logged in ASCII and may be displayed with the FUP COPY command. (See "Using Console Messages" in the System Operations Manual for your system.

3. An application process named $AOPR

   If an application process named $AOPR exists, all console messages are automatically logged to it. They are sent by means of an interprocess message (message -7). Messages sent to $AOPR contain the exact image of the console message. (See "Console Logging to an Application Process" in this section.)

Unlike user processes, the operator process does not open $AOPR prior to sending messages to it.

GENERAL CHARACTERISTICS OF THE OPERATOR CONSOLE

The operator console is accessed by $0 (dollar zero). The name of the operator console device is "$0 " (dollar sign, zero, followed by 22 blanks). It is opened like any other file (nowait opens are allowed).

Operator messages preempt terminal reads on the operator console. The operator console is considered by the operator process to be a write-only device.

Maximum message length for NonStop systems is 102 characters. This text is appended to the 29-character timestamp, for a total of 132 bytes. If the text exceeds 102 characters, any excess is lost. $0 reads a maximum of 102 characters, appends this data to the timestamp, and writes it to the console device and to OPRLOG and $AOPR (if enabled). The text is "folded" if the console device record length is less than (29 + text length), in which case the second line is indented 29 bytes.

The operator console device type is 1.

The file system automatically tries all available paths to the operator process; therefore, no path error recovery in the application program is required.

The size of the message queue for the operator process can be increased by BINDER. The default size of the message queue is unchanged from previous releases.

SUMMARY OF APPLICABLE PROCEDURES

The following procedures are used to write messages on the operator console:

DEVICEINFO	provides the device type and record length for the operator console.
OPEN	establishes communication with the operator process.
WRITE	is used to write a message on the operator console.

AWAITIO	waits for completion of an outstanding I/O operation pending to the operator process.
CANCELREQ	cancels the oldest outstanding operation, optionally identified by a tag, to the operator process.
FILEINFO	provides error information and characteristics about the operator process.
CLOSE	stops access to the operator process.

WRITING A MESSAGE

First the $0 file must be opened to provide access to the operator console:

```
   INT .OPERATOR [0:11] := ["$0", 11 * ["  "]],    ! data
        OP^FILE^NUM,                               ! declarations.
        MESSAGE[0:14];                             !
                                                   !
   STRING                                          !
       .SMESSAGE := @MESSAGE '<<' 1,               !
       .S;                                         !
      .
   CALL OPEN ( OPERATOR, OP^FILE^NUM );
      .
```

Then to send a message, use the WRITE procedure:

```
      .
   SMESSAGE ':=' "LOAD TAPE NO. 12345 ON UNIT 2" -> @S;
   CALL WRITE ( OP^FILE^NUM, MESSAGE, @S '-' @SMESSAGE );
```

The message appears on the console as follows:

```
  16:30 07SEP84 FROM 4,24 LOAD TAPE NO. 12345 ON UNIT 2
```

To terminate access to the operator console, use the CLOSE procedure:

```
      .
   CALL CLOSE ( OP^FILE^NUM );
      .
```

CONSOLE MESSAGE FORMAT

The general form of console messages is:

```
<timestamp> FROM <cpu>,<pin> <message>

<timestamp> 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 (those 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 generated and application generated) are logged to an application process named $AOPR if the process exists. Console message logging starts automatically upon detection of $AOPR. If console message logging is interrupted by the loss of $AOPR, it is restarted once $AOPR is reestablished.

The message is sent by the operator process as an interprocess message. The application process reads the message from its $RECEIVE file (see Section 4).

The operator process allows $AOPR 30 seconds to read this message. If it is not read, then an error message is logged on the console and to the disc log file.

The integer form of the console message to $AOPR is:

```
<opmsg>                 = -7
<opmsg>[1] FOR n        = ASCII text exactly as written to the
                          console (including BEL character on
                          some messages)

   n = the number of bytes in the console terminal form of
       the message.
```

NOTE

This is a user message, not a system message; a condition code of CCE is returned upon completion of the read.

SECTION 11

PROVIDING FAULT TOLERANCE WITH THE TRANSACTION MONITORING FACILITY (TMF)

The Transaction Monitoring Facility (TMF) is a software product that makes it easier for application programmers to protect the integrity of their data base in the face of hardware failure, software failure, and other complications. It performs failure-recovery operations that otherwise would have to be built into the application software.

Tandem systems are designed to minimize the problem of system failure in an online transaction-processing environment. It is a multiple-processor system, designed so that no single component failure can stop the processing.

Application processes typically run in pairs in separate processors so that the backup process can take over should the primary process's processor fail. Programming the backup processes and handling the checkpointing required to ensure that the backup processes are ready at all times, however, can be quite complicated. TMF does all this and more.

TMF uses audit trails, online dumps, and backout, rollforward, and autorollback facilities to ensure data base consistency even through a total system failure. If a transaction is aborted, all effects of that transaction are removed from the data base. After a system failure, TMF can remove all effects of any transaction that was interrupted. The other major service of TMF is transaction concurrency control. When you use TMF, the task of maintaining data consistency for a distributed data base that is being updated by concurrent transactions is simplified.

All the features of TMF are fully described in the Transaction Monitoring Facility (TMF) Reference Manual and the Transaction Monitoring Facility (TMF) System Management and Operations Guide.

PROGRAMMING FOR TMF

The general environment for applications using TMF is a requester-server environment where the requester accepts input from an operator and transforms it into a request for data base services from servers. The servers, in turn, satisfy the request by reading, locking, and changing (or adding or deleting) records in audited data base files. The requester can be written in SCREEN COBOL, TAL, FORTRAN, or COBOL. The server can be written in TAL, FORTRAN, or COBOL, and it must follow the record locking rules imposed by TMF. Only TAL applications are considered here.

Applications That Can Use TMF

Applications you intend to use with TMF should have the following characteristics:

- One process (the requester generally) coordinates all of the work required to do a single transaction. This process identifies the beginning and ending points of each TMF transaction. If the server replies to a request message by indicating that it failed to complete all of its changes, this process can decide to abort the transaction and abandon it, or abort the transaction and retry it.

- Communication between requesters and servers is by standard interprocess I/O. The requester does the WRITEREAD, and the server does the READUPDATE $RECEIVE and REPLY. Each request message and its reply is for a single transaction.

- Any disc I/O request is for a single transaction. TMF appends the process's current-transaction identifier to each disc-request message so the audit trails can include the identity of the transaction responsible for each data base change. This means that servers should not defer or anticipate an I/O call, and that a single I/O call should not combine work for more than one transaction.

- Any concurrency control is done by using the ENSCRIBE record locking facilities, and all servers should follow the record locking rules imposed by TMF. ENSCRIBE record locking gives TMF the control it requires to ensure that transactions are presented with a consistent view of the data base.

- Servers do not reply to request messages until all work for the request has been completed; the contents of the reply message should indicate whether the work for the request was completed successfully or abandoned in a partial state. This

characteristic lets the requester decide if the transaction should be committed (completed) or aborted.

- Servers should not employ checkpointing; this is unnecessary overhead with TMF and requires additional programming effort.

- Servers always perform all of their I/O for the request message most recently read from $RECEIVE and always reply to that message before reading another message; therefore, servers generally should not do $RECEIVE queuing. However, TMF does not forbid this practice; you can use $RECEIVE queuing, although it is considerably more difficult.

Defining the Transaction Identifier

In a Tandem system with TMF, each transaction is a uniquely identified entity. Each transaction is distinguished from other transactions by a four-word transaction identifier, which is created by a successful call to the BEGINTRANSACTION procedure.

The form of the transaction identifier is:

transid[0].<0:7>	contains 1 plus the EXPAND system number of the system in which BEGINTRANSACTION was called. It is 1 in the nonnetwork case. The system number identifies the home node of the transaction.
transid[0].<8:15>	contains the number of the processor in which BEGINTRANSACTION was called.
transid[1:2]	contains a doubleword sequence number that identifies the transaction.
transid[3]	contains a crash count indicating the number of times the home node (of the transaction) has had a total system failure since the last time the TMFCOM command INITIALIZE TMF was issued on the home node.

Because processes do the work for a transaction, the process control block (PCB) includes space for the identity of the transaction that is currently active for the process. The process's current-transaction identifier uniquely identifies the active transaction for a process. The phrase "restores to currency," as used in the rest of this section, means that a transaction identifier becomes the current-transaction identifier for a process.

TAL PROGRAMMING

You can use TMF with TAL applications by using these callable procedures:

- ABORTTRANSACTION aborts a transaction.
- ACTIVATERECEIVETRANSID is used to program $RECEIVE queuing servers; it restores the transaction identifier associated with a queued message request (for which REPLY has not been executed) that was previously acquired by reading $RECEIVE.
- BEGINTRANSACTION causes TMF to create a new transaction identifier.
- ENDTRANSACTION causes the data base changes associated with a transaction identifier to be committed.
- GETTMPNAME obtains the dummy device name of the transaction monitor process (TMP). (See "Using the Transaction Pseudofile (TFILE)" later in this section.)
- GETTRANSID obtains the transaction identifier of the calling process.
- RESUMETRANSACTION is used to program checkpointed requester processes or requester processes that have multiple concurrent active transactions; it restores a transaction identifier associated with a previous call to BEGINTRANSACTION.

PROGRAMMING CONSIDERATIONS

The following general considerations are related to programming for use with TMF:

- Accessing audited data base files
- Record locking
- Coding servers
- Avoiding deadlock
- Using the transaction pseudofile (TFILE)
- Handling TMF backout anomalies

Accessing Audited Data Base Files

TMF guarantees the consistency of data through the use of audited files. Audited files are files that have been so designated by the use of FUP or the CREATE procedure. Audit trails are written to disc before a transaction commits its changes. Before-images and after-images of all changes to audited files are written to the audit trails. After-images are used by rollforward and autorollback to redo changes made by committed transactions. Before-images are used by rollforward, backout, and autorollback to udo changes made by aborted or incomplete transactions.

Figure 11-1 illustrates the differences between processes that change audited files and those that can change only nonaudited files.

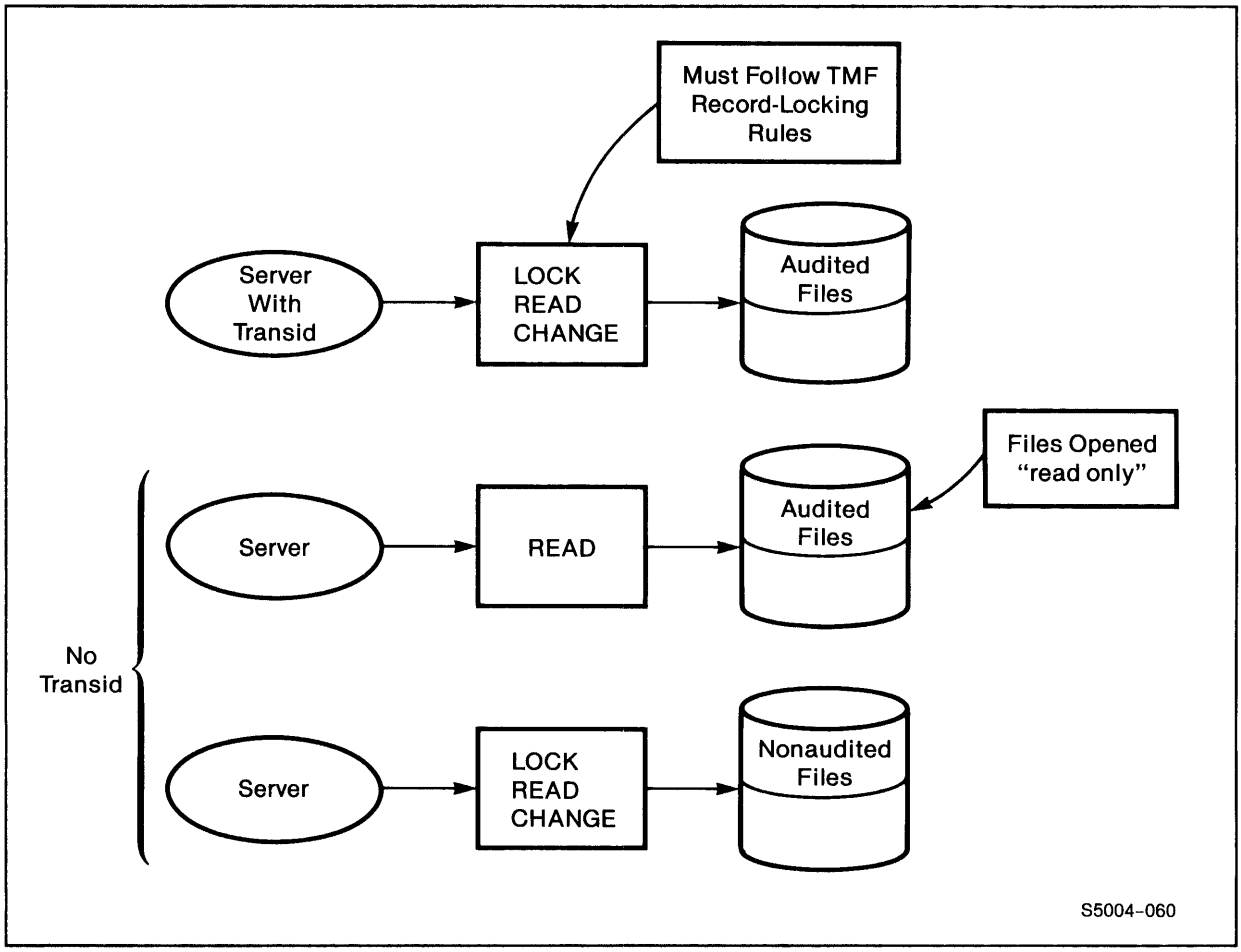


Figure 11-1. Accessing and Changing Audited as Opposed to Nonaudited Files

In a Tandem system with TMF, each transaction must have a transaction identifier. A process without a transaction identifier can read records in audited files, but it cannot lock or change them.

A transaction identifier is created when a requester calls BEGINTRANSACTION. A server automatically acquires a transaction identifier when it reads $RECEIVE to pick up a request message sent by a requester that has a current transaction identifier. If a server has a transaction identifier, it can read, lock, insert, delete, and change records in audited files.

A process generally acquires the transaction identifier when the server reads $RECEIVE. However, when a process initiates a transaction (the requester calls BEGINTRANSACTION) and thus acquires a current-transaction identifier, the identifier associated with the requester is not replaced by one associated with the $RECEIVE message.

Record Locking

For all changes to audited files, TMF enforces the following record locking protocol:

- An existing record must be locked by a transaction before it can be changed or deleted by a transaction.
- TMF locks all records inserted by a transaction.
- TMF locks the primary keys of all records deleted by a transaction. This ensures that the record can be reinserted if the transaction aborts.
- TMF will not release the locks for any record changed, inserted, or deleted by a transaction until the transaction either is committed or aborts and is backed out.

Locks can be acquired individually on a record-by-record basis or a lock can be acquired for an entire file by using the GUARDIAN LOCKFILE procedure. Figure 11-2 illustrates (1) how processes can acquire locks and update audited files and (2) when TMF will release the locks.

If the whole set of current active transactions tries to acquire more than 2000 key locks or 3000 record locks per file, sufficient extended segment space might not be available.

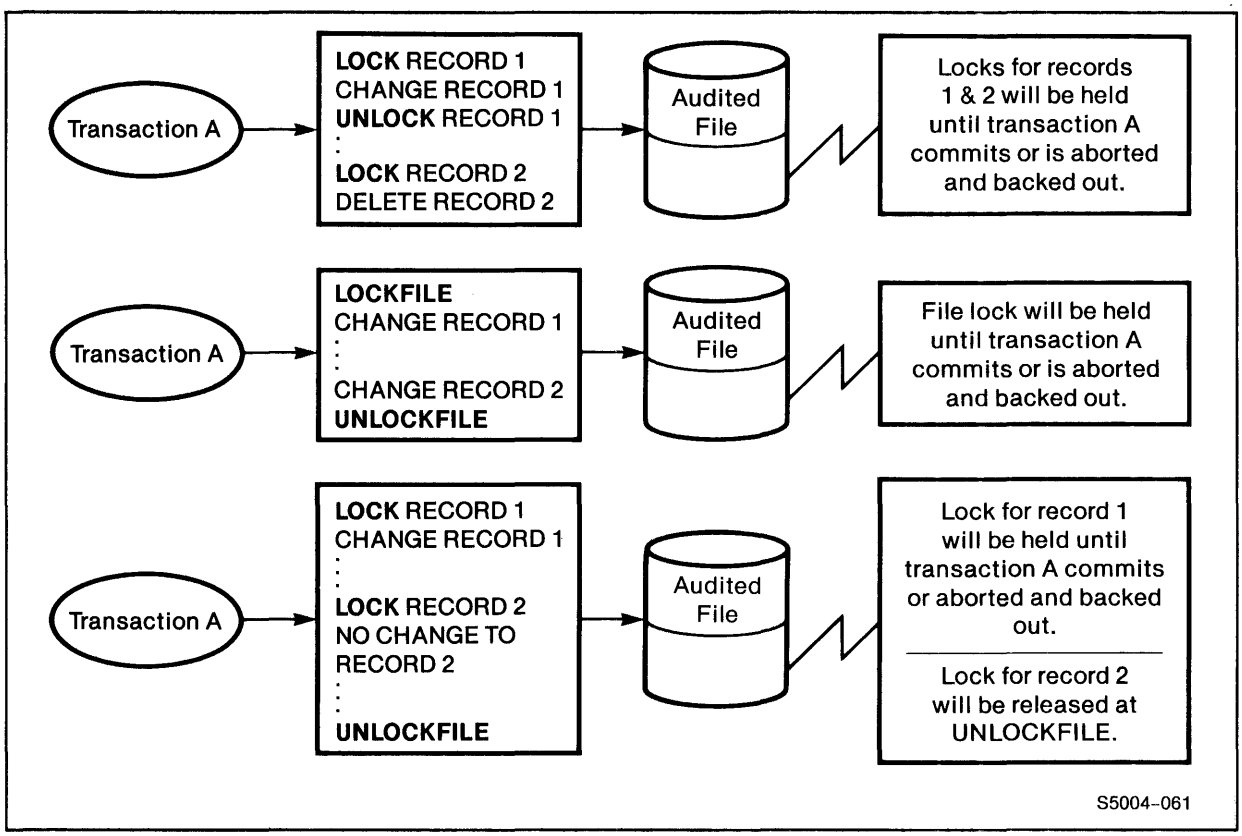


Figure 11-2. Record Locking for TMF

The file lock or record locks are owned by the current-transaction identifier of the process that issued the lock request. In PATHWAY, for example, a single transaction can send requests to several servers or multiple requests to the same server class. In this situation, where several processes share a common transaction identifier and the locks are held by the same transaction identifier, the locks do not cause conflict among the processes participating in the transaction (see Figure 11-3).

Figure 11-3 illustrates the following principles:

- The terminal control process (TCP), which functions as a multithreaded requester, interprets BEGINTRANSACTION and obtains the transaction identifier before requesting data base activity from the servers.

- The transaction identifier is transmitted to the servers in the request message, and any disc activity performed by the servers is associated with the transaction identifier.

- The transaction identifier owns the locks; all servers that acquired the same transaction identifier can read, lock, add, delete, and change records in the audited files. For example, server A can read and lock a record, and server B can read or change the same record if both servers A and B have the same transaction identifier.

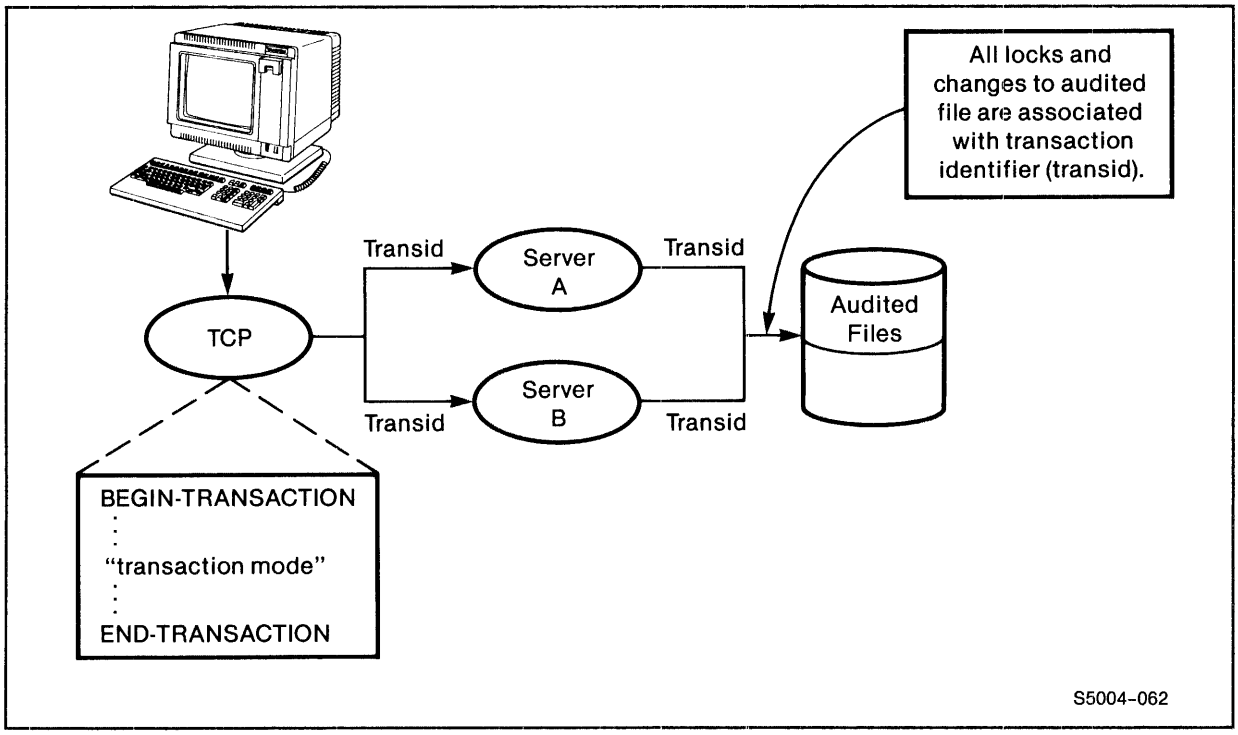


Figure 11-3. Record Locking by Transaction Identifier

Repeatable Reads

Generally, a transaction should lock any data it reads and uses in producing its output, regardless of whether it modifies the data. Following this rule guarantees that all of a transaction's reading operations are repeatable and that data on which the transaction depends does not change before the transaction is committed.

Opening Audited Files--Errors

Because locks are owned by the transaction identifier instead of the process identifier or the identifier of the file opener, they can persist longer than the opener process. This means that even if a file has been closed by all its openers, the disc process keeps it effectively open until all transactions owning locks in the file have ended or have been aborted and backed out.

For files with pending transaction locks, these types of errors are possible:

- Attempting to open an audited file with exclusive access fails with file-system error 12 (file in use), regardless of whether openers of the file exist.
- FUP operations requiring exclusive access such as PURGE and PURGEDATA fail. PURGE fails with file-system error 12 and PURGEDATA fails with file-system error 80.

Additionally, file-system error 80 (invalid operation on audited file) is returned for these OPEN situations:

- Attempting to open an audited file having an automatically updated alternate key file that cannot be opened or is not audited
- Attempting to open a structured audited file with unstructured access
- Attempting to open an audited, partitioned file having a nonaudited secondary partition.

Reading Deleted Records

If transaction T1 deletes a record, and another transaction T2 attempts to read the same record while T1 is still active, two possible errors could result:

- If the read request is the GUARDIAN procedure READ after exact positioning, then file-system error 1 (end of file) is returned.
- If the read request is the GUARDIAN procedure READUPDATE, then file-system error 73 (file/record locked) is returned in alternate locking mode, and the request waits for T1 to complete in default locking mode.

Batch Updates

When programming for batch updating of audited files, you should either have the transaction lock an entire file at a time by using the LOCKFILE procedure or carefully keep track of the number of locks held. If you do not use LOCKFILE, TMF sets these implicit locks:

- When a new record is inserted in an audited file, TMF implicitly locks that record.
- When a record is deleted from an audited file, TMF implicitly locks the key of that record.

These locks are not released until the transaction is committed or is aborted and backed out. This means that transactions doing batch updates to audited files, if they involve deleting or inserting a large number of records, can obtain too many locks. (The maximum number of locks that can be acquired for each file is approximately 2000 key locks and 3000 record locks.)

If a transaction calls LOCKFILE for a primary-key file, LOCKFILE is also applied to any associated alternate-key files. This prevents primary-file updates from causing the alternate-key files to obtain record locks (for insertions) or key locks (for deletions).

Coding Servers

Figure 11-4 illustrates the typical sequence of actions performed by a single-threaded (not $RECEIVE queuing) server.

When you write servers of the type illustrated in Figure 11-4, consider the following:

- When the server reads $RECEIVE to pick up its request message, it automatically acquires the transaction identifier of the process that sent it the message. All data base operations performed from the point of the read on $RECEIVE until the server replies are associated with the transaction identifier.

- Existing servers that (1) are not checkpointed, (2) do not do $RECEIVE queuing, and (3) lock all records before changing them generally do not have to be modified for TMF.
- The server must follow the record locking rules imposed by TMF. This means it must lock all records that it deletes or changes. In addition, if it requires repeatable reads, it should lock records that it reads and uses in producing its output, even if it does not modify the records.

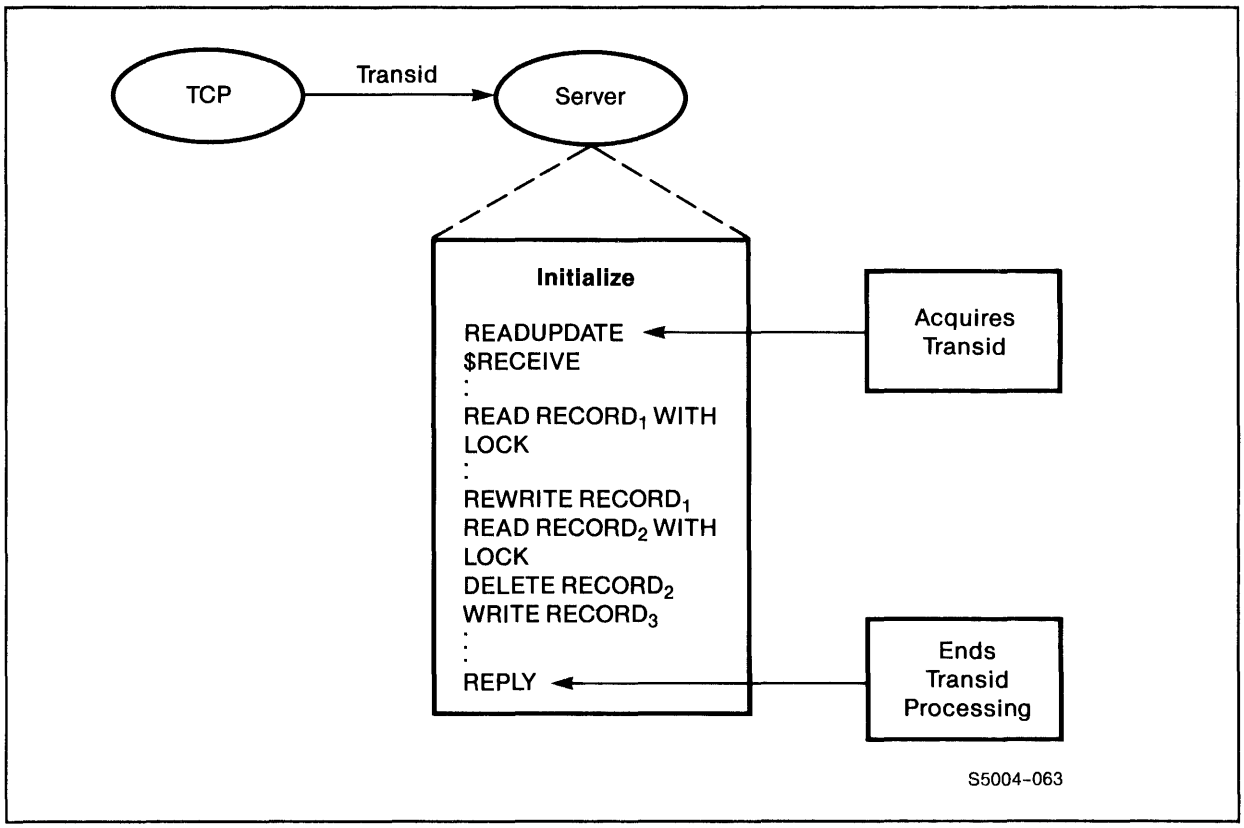


Figure 11-4. Nonqueuing Server

Figure 11-5 illustrates a typical sequence of actions performed by a $RECEIVE queuing server.

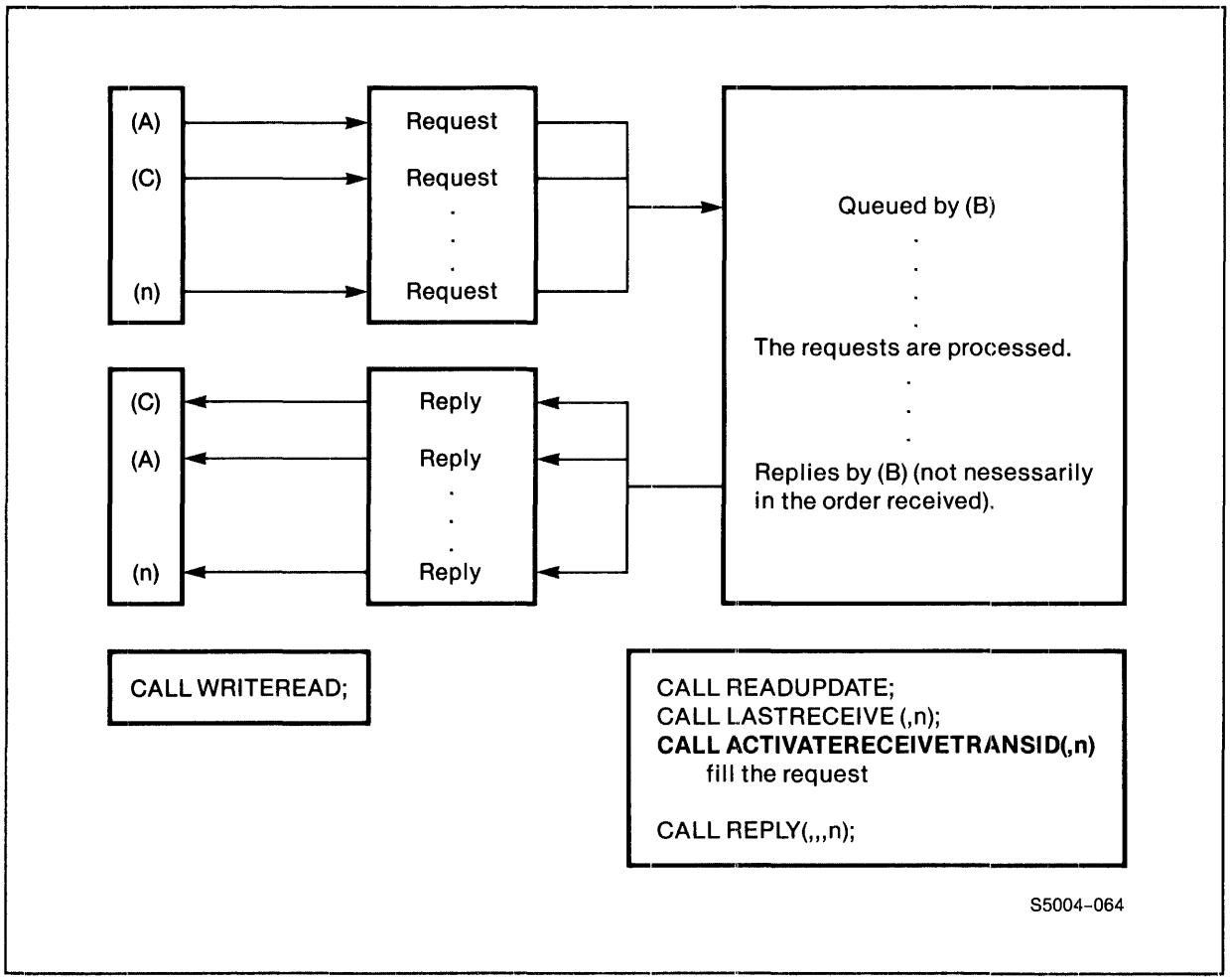


Figure 11-5. $RECEIVE Queuing

A server of the type illustrated in Figure 11-5 identifies the requester associated with the message by obtaining its message tag through a call to the GUARDIAN procedure LASTRECEIVE or RECEIVEINFO. It can then indicate which message it is responding to by specifying the message tag as a parameter in REPLY. Since a $RECEIVE queuing server does several READUPATEs on $RECEIVE before issuing REPLYs, it needs to acquire the current transaction identifier dynamically. That is, whenever it does some operations for a request message, it must assume its transaction identifier for the duration of the operations and then acquire the transaction identifier of the next message it is to work on. A call to ACTIVATERECEIVETRANSID with the message tag returned by an earlier call to LASTRECEIVE or RECEIVEINFO lets the server specify that the transaction identifier of the message associated with the message tag should become current for the process.

Avoiding Deadlock

The following example of a sequence of record locking operations results in a deadlock situation:

1. Transaction 1 locks record A.
2. Transaction 2 locks record B.
3. Transaction 1 attempts to lock record B and has to wait.
4. Transaction 2 attempts to lock record A and has to wait.

Neither transaction can proceed, and the situation is a deadlock.

Some deadlock situations that can occur because of the record locking protocol of TMF are:

- Deleting a record implicitly locks the key of the record and can cause the deadlock situation illustrated in Figure 11-6.

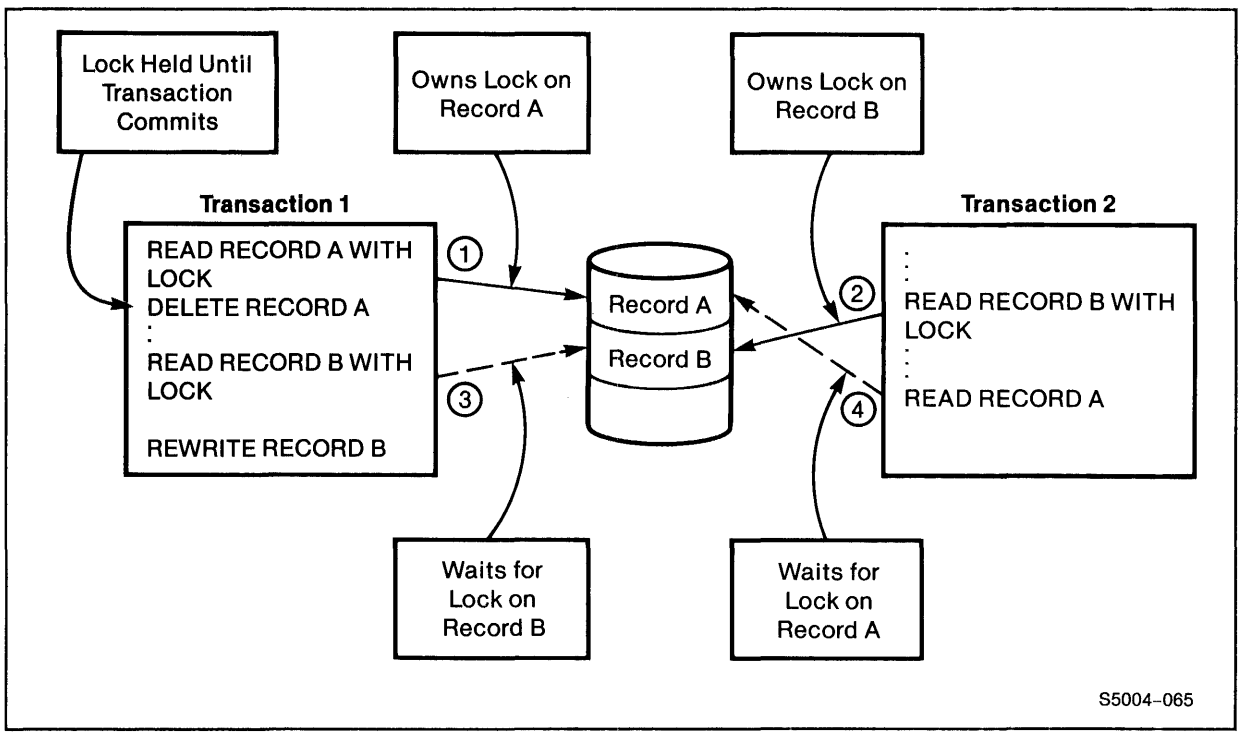


Figure 11-6. Deadlock Caused by Deleting a Record

- A record inserted by a transaction is automatically locked and can cause the deadlock situation illustrated in Figure 11-7.

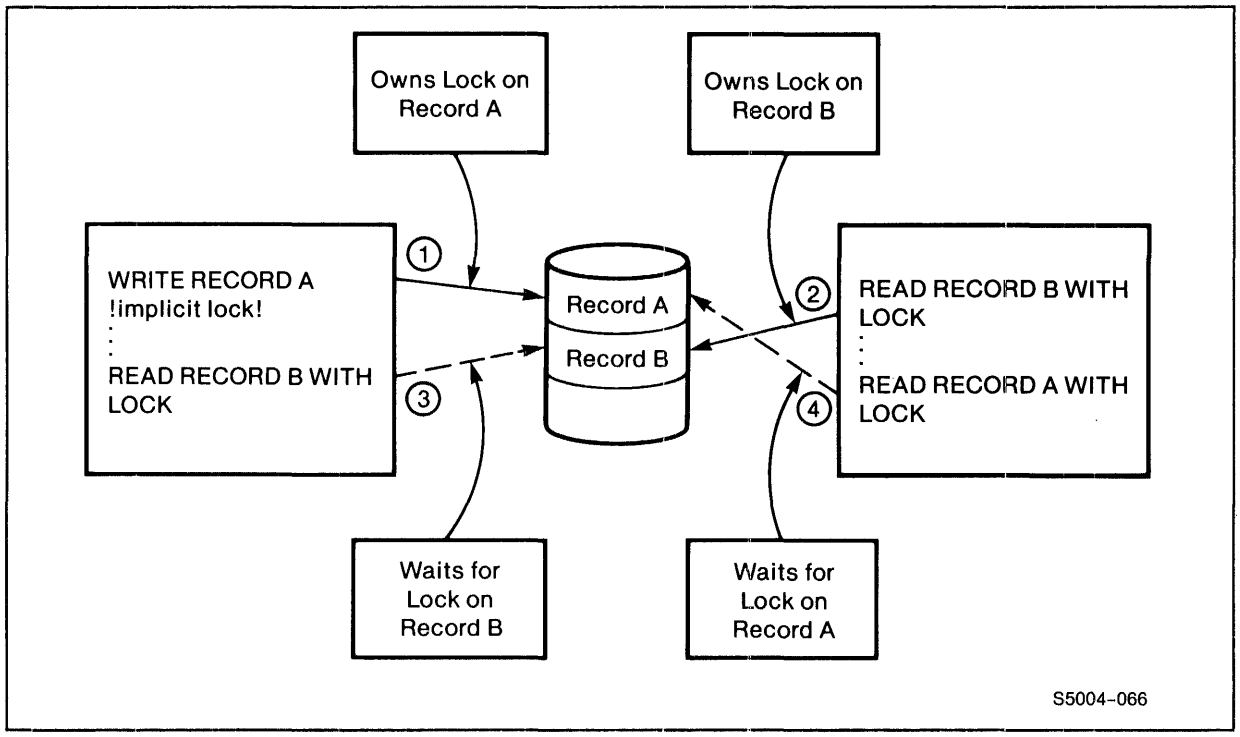


Figure 11-7. Deadlock Caused by Inserting a Record

- A process can deadlock itself, as illustrated in Figure 11-8, if it acquires different current-transaction identifiers.

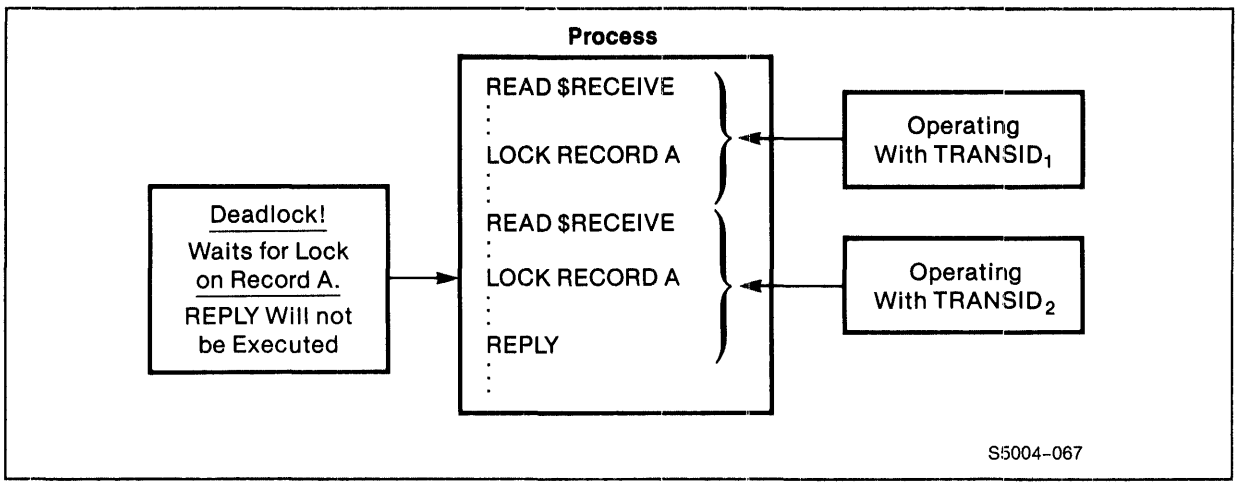


Figure 11-8. Deadlock Caused by a Process Switching Transaction Identifiers

Multiple SENDs to one PATHWAY server, if they cause it to access the same record under a different transaction identifier, can cause the server to participate in a deadlock (Figure 11-9). This situation occurs only if different terminal control processes (TCPs) are involved in the SENDs.

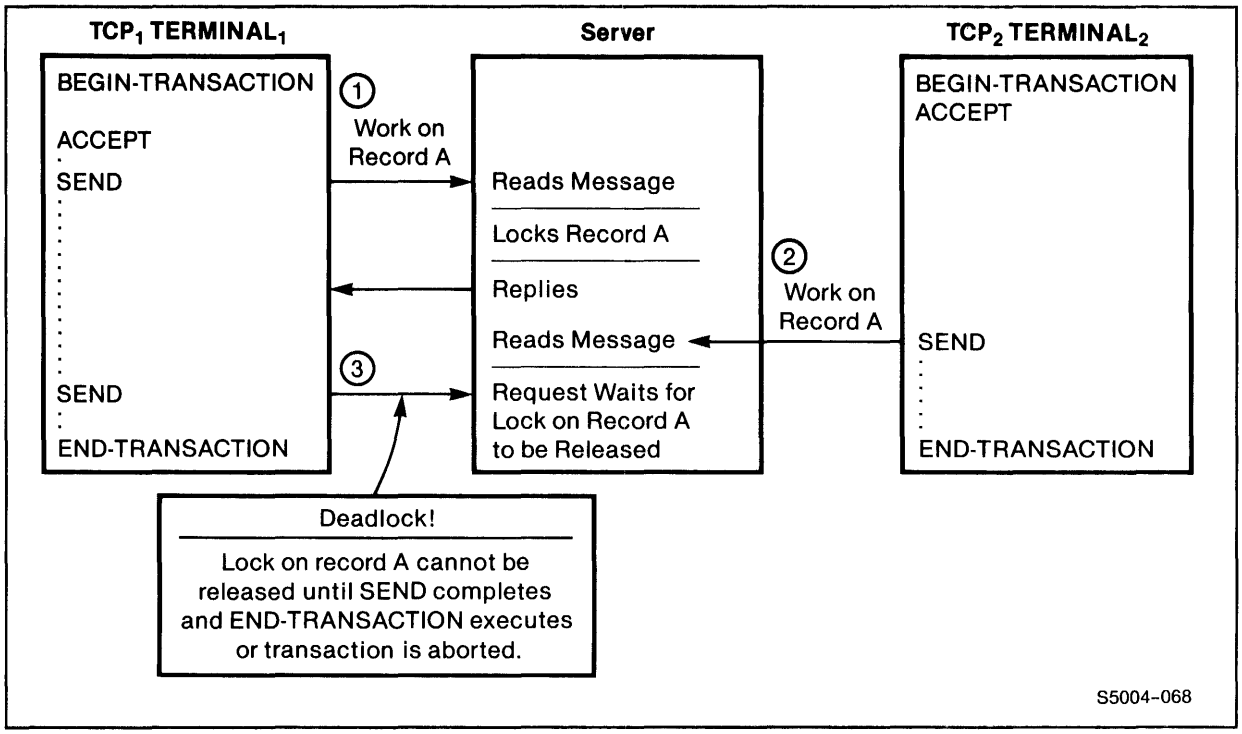


Figure 11-9. Deadlock Caused by Multiple SENDs

There is no way to detect if a transaction becomes involved in a deadlock. However, the following situations can be detected:

- A transaction is attempting to read or lock a record that is is already locked.
- A transaction read or lock request is waiting too long before completion.

Each of these situations is explained below and illustrated in Figure 11-10. In either situation, it is safe to assume (though it may not be true) that the transaction is in a deadlock and to program the transaction to abort or restart. The locks held for the transaction are released, avoiding the possibility of it participating in or prolonging a deadlock. For PATHWAY, the server can return a message to the requester that indicates the deadlock possibility, and the requester can respond, for example, with the COBOL ABORT-TRANSACTION or RESTART-TRANSACTION verb.

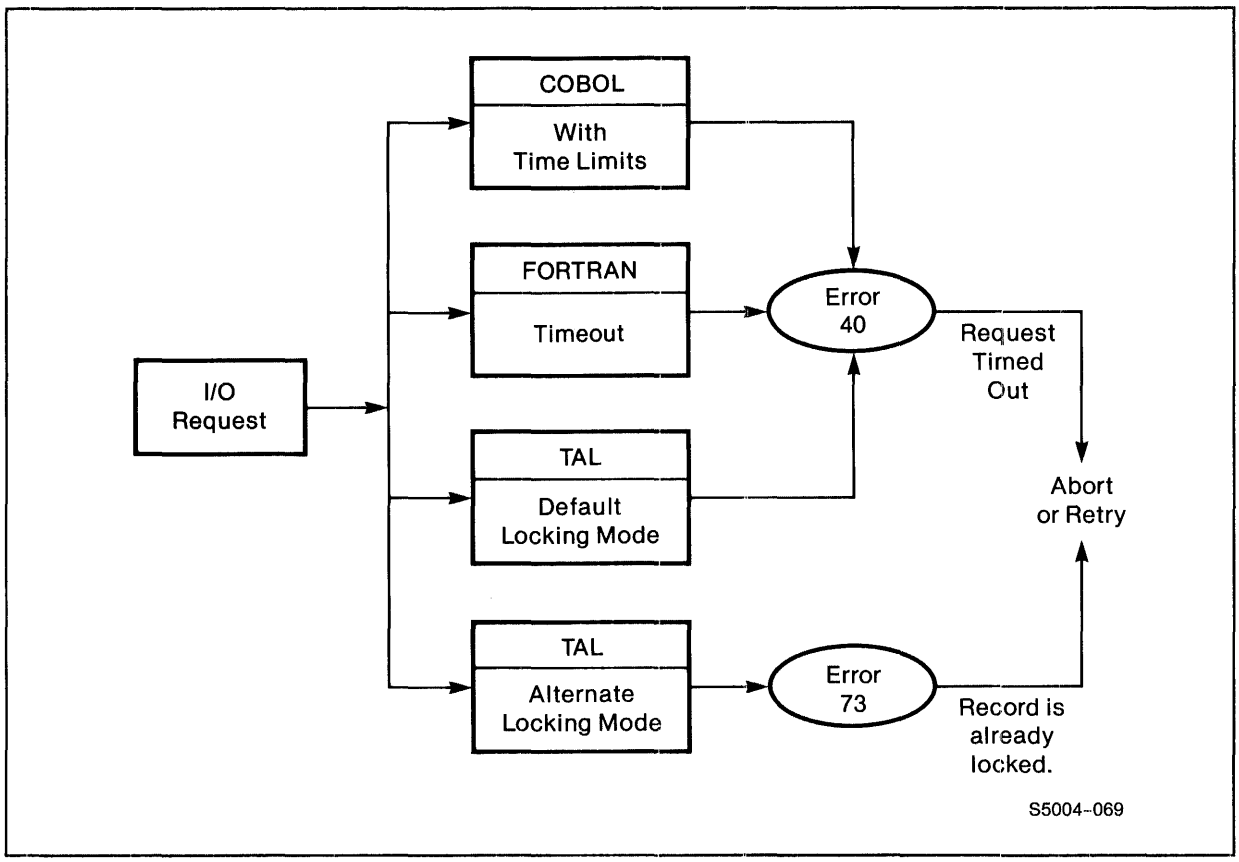


Figure 11-10. Avoiding Deadlock

TAL programmers can determine if a record is already locked by using the GUARDIAN SETMODE procedure to select alternate locking mode. In this mode, file-system error 73 is returned to the request when it attempts to access a locked record.

In default locking mode, TAL programmers can determine if an I/O request has waited too long before completion. In this mode, a process will be suspended when it attempts to access a locked record. To avoid deadlock, open the file using nowait I/O and specify a nonzero time limit in the call to AWAITIO. If AWAITIO returns file-system error 40 (indicating timeout), the transaction may be in a deadlock situation.

COBOL programmers can open files using the WITH TIME LIMITS parameter. WITH TIME LIMITS indicates that further I/O requests will be timed by specifying a value in the TIME LIMIT parameter of the request. If the I/O request times out, file-system error 40 is returned to the request.

FORTRAN programmers can open files with the TIMED specifier and use the TIMEOUT specifier in their I/O requests to specify a timeout value. If the I/O request times out, file-system error 40 is returned to the request.

Using the Transaction Pseudofile (TFILE)

The transaction pseudofile (TFILE) is not a physical I/O device file; it is never the target of actual I/O operations. Instead, it provides control block space (an ACB) for these types of processes:

- A checkpointed process that executes BEGINTRANSACTION where the backup does transaction recovery on takeover
- A process that requires nowait calls on ENDTRANSACTION because it executes multiple BEGINTRANSACTION calls without intervening ABORTTRANSACTION or ENDTRANSACTION calls
- A process which, after calling BEGINTRANSACTION, interrupts the transaction to work on a different transaction, then calls RESUMETRANSACTION to continue working on the first one

The TFILE is necessary because these processes can begin multiple transactions; the TFILE ACB provides space to store the history of each transaction's completion status (aborted or ended) until the transaction is completed and individual transaction results have been returned to the process.

When a process opens the TFILE with nonzero sync depth, ENDTRANSACTION is always executed on a nowait basis and must be completed with a call to the AWAITIO procedure. However, if the TFILE is opened with depth zero, ENDTRANSACTION is a waited operation and AWAITIO is unnecessary. If the process does not open a TFILE, its BEGINTRANSACTION call opens one for it (with a depth of zero) and the TFILE remains open until the process stops.

Whenever a process executes BEGINTRANSACTION successfully, a new transaction request entry is placed in the TFILE for the calling process (primary process only in a checkpointed process pair). The TFILE transaction request entry is deleted when ENDTRANSACTION is called, and the completion status of the transaction has been returned from AWAITIO or when ABORTTRANSACTION is called.

For a transaction whose completion status is "ended", the transaction is considered complete upon the return from AWAITIO on the TFILE for the associated transaction identifier tag.

For a transaction that is voluntarily aborted, its completion is signalled by the return of the ABORTTRANSACTION call.

Opening the TFILE

The TFILE is opened using the symbolic name of the TMP's logical device name. You should obtain this name programmatically by calling the GETTMPNAME procedure; the name is normally $TMP.

GETTMPNAME returns a zero if it succeeds, or one of these file-system error numbers:

- 22--Parameter is out of bounds.
- 84--TMF is not configured.

A process can open the TFILE only once; attempts to open multiple instances of the TFILE or to open the TFILE after BEGINTRANSACTION has once been called, fail with file-system error 12.

The OPEN parameters required to open the TFILE are <filename>, <filenum>, and <flags>. The <flags> parameter specifies the maximum number of transactions that the process can have concurrently active, and it must be between 1 and 100, inclusive. If <flags> is greater than 100, the attempt to open the TFILE will fail with file-system error 28. Using <flags> is analogous to using the nowait depth for a standard file, except that its maximum value is 100 rather than 15.

Using AWAITIO to Complete ENDTRANSACTION Calls

When using the TFILE, ENDTRANSACTION calls are treated as nowait operations and must be completed with calls to AWAITIO. A checkpointed process that executes BEGINTRANSACTION and does transaction recovery on takeover must complete its ENDTRANSACTION call by calling AWAITIO, even if it never has more than one transaction in process at any one time.

AWAITIO tests for, or waits for completion of, an ENDTRANSACTION call. AWAITIO is used for the TFILE in the same manner as it is for standard files. The <filenum> parameter means the same thing. However, the BEGINTRANSACTION tag of the ended transaction is returned through the tag parameter of AWAITIO; this is the same tag that is used in RESUMETRANSACTION.

Synchronizing the TFILE ACBs

The TFILE ACBs of checkpointed process pairs must be synchronized by using checkpoints to allow correct transaction recovery during takeover. The TFILE should be checkpointed after the situations described below; it performs different functions in each of the situations.

In the first situation, the last precheckpoint operation performed by the primary process is either BEGINTRANSACTION or RESUMETRANSACTION. Checkpointing the TFILE adds (or verifies the presence of) a transaction request in the backup process's TFILE ACB that corresponds to the primary process's current-transaction identifier at the time of the checkpoint.

In the second situation, the last precheckpoint operation performed by the primary process is either TFILE AWAITIO or ABORTTRANSACTION. Checkpointing the TFILE deletes the transaction request from the backup process's TFILE ACB that corresponds to (1) the transaction identifier for the completion status that was most recently returned by AWAITIO on the TFILE, or (2) the transaction identifier for which ABORTTRANSACTION was called.

Using the TFILE for Checkpointed Operations

For BEGINTRANSACTION processes that use the TFILE, creating and bringing up a backup process requires that the primary process:

- CHECKOPEN the TFILE
- Call RESUMETRANSACTION for each transaction identifier tag that corresponds to a transaction that the backup process must recover in the event of a takeover.

If you have created a backup process, do not checkpoint the TFILE to it while any transactions are ending (that is, when ENDTRANSACTION has been called, but the corresponding AWAITIO has not yet completed successfully).

Handling TMF Backout Anomalies

When a transaction aborts, TMF backs it out as follows:

- Records updated by the transaction are backed out by a WRITEUPDATE of the before-images for each of the updated records.
- Records deleted by the transaction are backed out by a WRITE (insert) of the before-images for each of the deleted records.
- Records inserted by the transaction are backed out by a write-count zero WRITEUPDATE (delete).

The following anomalies can occur during backout:

- Insertions at end of file (EOF) to an unstructured file cannot be backed out. This means that EOF will not be restored to its previous value, because another transaction may have written to EOF after the insert but before the backout.
- If a record is inserted at EOF in an entry-sequenced file, the record is backed out by rewriting it with a length of zero bytes; that is, making it an empty record (even if at EOF). A READ at that record's address then will return a null record with a length of zero bytes.
- An EOF (-1D POSITION) insertion to a relative file will be backed out by deleting the record. However, EOF will not be restored to some previous value because another transaction may have written to EOF after the insert but before the backout.
- Backout can fail for a transaction that deletes records from a key-sequenced file that is near a "file full" condition. This occurs if other transactions, concurrently with the transaction that deleted the record, insert enough records to fill the file. If the file is full, the transaction that deleted the records cannot be backed out because there is insufficient space to insert the records that it deleted. If this happens, the console will get a message similar to the following:

  ```
  TMF Backout : error 45 at location <loc> - undo^error
  TMF Backout : file involved in above error - <filename>
  TMF Backout : transid involved in above error - <transid>
  ```

  and the transaction to delete the record will remain in an aborting state. Note that the message reflects file-system error 45, "file is full." If your application programs maintain a log to tie the transaction identifier to the user-entered transaction, you may be able to correct the

problem manually. It may also be possible to run a separate transaction to delete one or more unneeded records, then ask the operator to use the TMFCOM ABORT TRANSACTION command to complete the previously unsuccessful backout. (A better solution for this problem is to use the FUP INFO command periodically and make sure your files never get that full.)

Advanced Usage of TMF

Analysts and system programmers who wish to obtain further information on the use of TMF in conjunction with checkpointing, can refer to the Tandem Journal, Volume 1, Number 1, Fall 1983. It contains an article entitled "TMF and the Multi-Threaded Requester."

Checkpointing can be useful in conjunction with TMF. TMF can do everything that checkpointing can do, with one exception: TMF cannot reinitiate failed transactions.

Checkpointing can be used to make processes that perform TMF transactions "persistent". Persistent process pairs can survive single failures while continuing to perform transactions.

A persistent process can use a simple recovery strategy. If the primary process fails while processing a transaction, the backup process can rely on TMF to backout the incomplete transaction. The backup may either resubmit the failed transaction or go on to the next transaction. This is both simpler and more efficient than keeping the backup process constantly apprised of the transaction's state as it is being performed.

SECTION 12

WRITING FAULT-TOLERANT PROGRAMS

The term "fault tolerant" means that a single hardware component failure will not cause processing to stop; moreover, in most cases, a software problem will not cause processing to stop.

From the hardware side, fault tolerance is not difficult to achieve on a Tandem system. Sufficient redundant hardware and duplication of paths allow Tandem systems to tolerate a single-component failure. In many cases, multiple-component failures can also be tolerated as long as they do not share common paths. Moreover, the redundant paths are not duplicate backups; that is, all available resources are used for processing--none are held in reserve for use as spare backups. The hardware concepts employed to achieve this fault tolerance are explained in the Introduction to Tandem Computer Systems.

Tandem software is also fault tolerant. Many software problems are not repeatable; that is, the problem can occur only when a combination of events creates a given environment. Most other computer systems cannot deal with problems of this nature because their backup systems encounter the same bug and fail in the same manner as their primary system because the backup system shares the same environment. On a Tandem system, such software bugs usually do not halt processing because the combination of events that led to the failure is not repeated in the backup process.

There are several methods you can use to reduce the time required for recovery should your program fail. Of these methods, checkpointing is one of the most important. It provides a method of saving information at a given point in processing so that a process can be restarted at that point rather than having to return to the beginning of its execution.

Checkpointing is an integral part of the GUARDIAN operating system. It is commonly used in combination with other tools to support fault-tolerant processing.

Checkpointing is done to ensure that a backup process can take over from the primary process at the correct point. The primary process and the backup process together form an entity known as a NonStop process pair. The purpose of the checkpoint information is to enable the backup process to recover from a failure of the primary process in an orderly manner.

Checkpoints do not occur automatically. As a programmer, you must determine where you want to place the checkpoints within your program, and what information you want sent at each checkpoint. The placement of checkpoints in transaction processing is critical. Especially in the final phases of coding your program, you must constantly be alert to the possibility that errors can result if you fail to checkpoint all data that has been modified.

NOTE

If the Transaction Monitoring Facility (TMF) is available on your system, you can use TMF to avoid using checkpoints in many applications. TMF is described in Section 11.

CHECKPOINTING PROCEDURES

The checkpointing facility consists of a set of procedures that are used:

- To assume control in the backup process in case of failure of the primary process or its processor module:

  CHECKMONITOR (backup-process)

- To open and close a process pair's files:

  OPEN and CLOSE (primary process)
  CHECKOPEN and CHECKCLOSE (primary process)
  CHECKMONITOR (backup process)

- To checkpoint the execution state of a primary process to its backup process:

  CHECKPOINT or CHECKPOINTMANY (primary process)
  CHECKMONITOR (backup process)

- To transfer control to the backup process so that the system load is redistributed, and for testing fault tolerance:

  CHECKSWITCH (primary process)
  CHECKMONITOR (backup process)

- To request notification of a change in the operational state of one or more processor modules:

  MONITORCPUS (primary and backup processes)

- To obtain the count and operational states of processor modules:

  PROCESSORSTATUS

- To obtain the processor type of a specified system and CPU:

  PROCESSORTYPE

See the System Procedure Calls Reference Manual for details.

WHAT INFORMATION IS CHECKPOINTED?

The following types of information can be checkpointed:

- The process's data stack

  The data stack, in this context, is considered to be the area from an address specified in the call to CHECKPOINT (usually the address of the last global variable) through the current top-of-stack location (the word pointed to by the current setting of the S register). This area contains the local data storage for all currently active procedures and their stack markers.

- Individual blocks of data in the data area

  These are usually file buffers, but may be any data desired.

- Disc file "sync blocks"

  A sync block contains control information about the current state of a disc file (such as the current value of the file pointers).

When a call to CHECKPOINT is made by the primary process, a message containing the information to be checkpointed is formatted and sent to the backup process in the form of an interprocess message. The message is received and processed by the CHECKMONITOR procedure in the backup process.

Data Stack

The purposes of checkpointing the data stack are to provide a restart point for the backup process and to preserve the values of the process' variables at the time of the checkpoint. This is possible because the stack markers in the data stack define the executing environment of the primary process at the time of the call to CHECKPOINT, and because the primary's data stack is duplicated in the backup. If the primary process fails, CHECKMONITOR simply returns through the stack marker for the latest call to CHECKPOINT. In this manner, the backup begins executing following the latest call to CHECKPOINT.

Data Buffers

The purpose of checkpointing data buffers is to preserve the state of the process so that the backup can continue processing. Typically, data buffer checkpointing occurs just before writing to a disc file; the data about to be written is checkpointed. Careful selection of which data buffers (and corresponding file sync information, discussed in the following paragraphs) to checkpoint can increase the efficiency of a fault-tolerant program. An example of data buffer checkpointing is an entry received from a terminal; the data buffer is checkpointed to minimize the possibility that the operator would have to reenter data. Data buffers residing in the data stack are checkpointed when the stack is checkpointed.

Sync Blocks

The purpose of checkpointing the sync block is twofold:

1. To ensure that no write operation is duplicated when a backup takes over from its primary
2. To pass the current values of file pointers to the file system on the backup side

When a checkpoint of the sync block occurs, the information in the sync block is passed to the file system by CHECKMONITOR.

The need to prevent duplicate operations is illustrated in the following sequence:

A primary completes the following write operation successfully but fails before a subsequent checkpoint to its backup:

```
RESTART POINT ------>(C)   CHECKPOINT POSITION AND DATA
                      |
                     x  POSITION(F1,-1D); ! position to eof
                     x  WRITE(F1,F1^BUFFER);
                      |
            *** FAILURE OF PRIMARY ***
```

On the takeover from the primary, the backup reexecutes the operations just completed by the primary. If the WRITE were performed as requested, it would duplicate the record, but at the new end-of-file location.

To prevent a write operation already performed by the primary from being duplicated by the backup process, the <sync-depth> parameter of OPEN must be specified as a value greater than zero when opening the file. For a file opened in this manner, a sync ID in the sync block is used by the file system to identify the operation about to be performed by the backup in the event of a primary process failure. If the backup requests an operation already completed by the primary, the file system, through use of the sync ID, recognizes this condition. Then, instead of performing the requested operation, the file system returns the completion status of the operation to the backup (the completion status was saved by the file system when the primary performed the operation). However, if the requested operation has not been performed, it is performed and the completion status is returned to the backup. The course of action that is taken by the file system is completely invisible to the backup process.

The file system has the capability to save the completion status of the latest operations against a file and to relate those completions to operations requested by a backup process upon takeover from a failed primary process. The maximum number of completion statuses that the file system is to save is specified in the <sync-depth> parameter to OPEN. The <sync-depth> value is typically the same as the maximum number of write operations to a file without an intervening checkpoint of the file's sync block. In most cases, the <sync-depth> value is 1; it cannot exceed 15.

Information Not Checkpointed

Operations that are "retryable" usually are not checkpointed; they can be retried until the operation succeeds. Retryable operations are those that do not alter the data base, and can be reexecuted indefinitely with the same results, without

duplication or loss of data. Those operations which are not retryable should be checkpointed.

Operations that are not retryable cannot be repeated. Therefore, each request message contains a sync ID that is used to detect and negate duplicate requests for nonretryable operations. Refer to "File Synchronization Information" near the end of this section for more information.

OVERVIEW OF FAULT-TOLERANT TRANSACTION PROCESSING

Process pairs form the basis for fault-tolerant processing. The primary process and the backup process execute the same program file as illustrated in Figure 12-1.

```
Save stack base address;

CALL GETCRTPID (MYPID, PPDENTRY);  ! find out if this is the
                                   ! primary or backup
CALL LOOKUPPROCESSNAME (PPDENTRY);
IF < THEN CALL ABEND;              ! no name

IF PPDENTRY [4] THEN               ! this is the backup
   BEGIN
   CALL MONITORCPUS (PRIMARY^CPU);
   CALL CHECKMONITOR;
   CALL ABEND;
   END;
CALL MONITORCPUS (BACKUP^CPU);
  .
read startup message               ! this is the primary
  .
OPEN files
  .
create backup
  .
CHECKOPEN files
```

Figure 12-1. Sample Startup Sequence for a Process Pair

The actions of the primary and backup processes are shown in Figure 12-2.

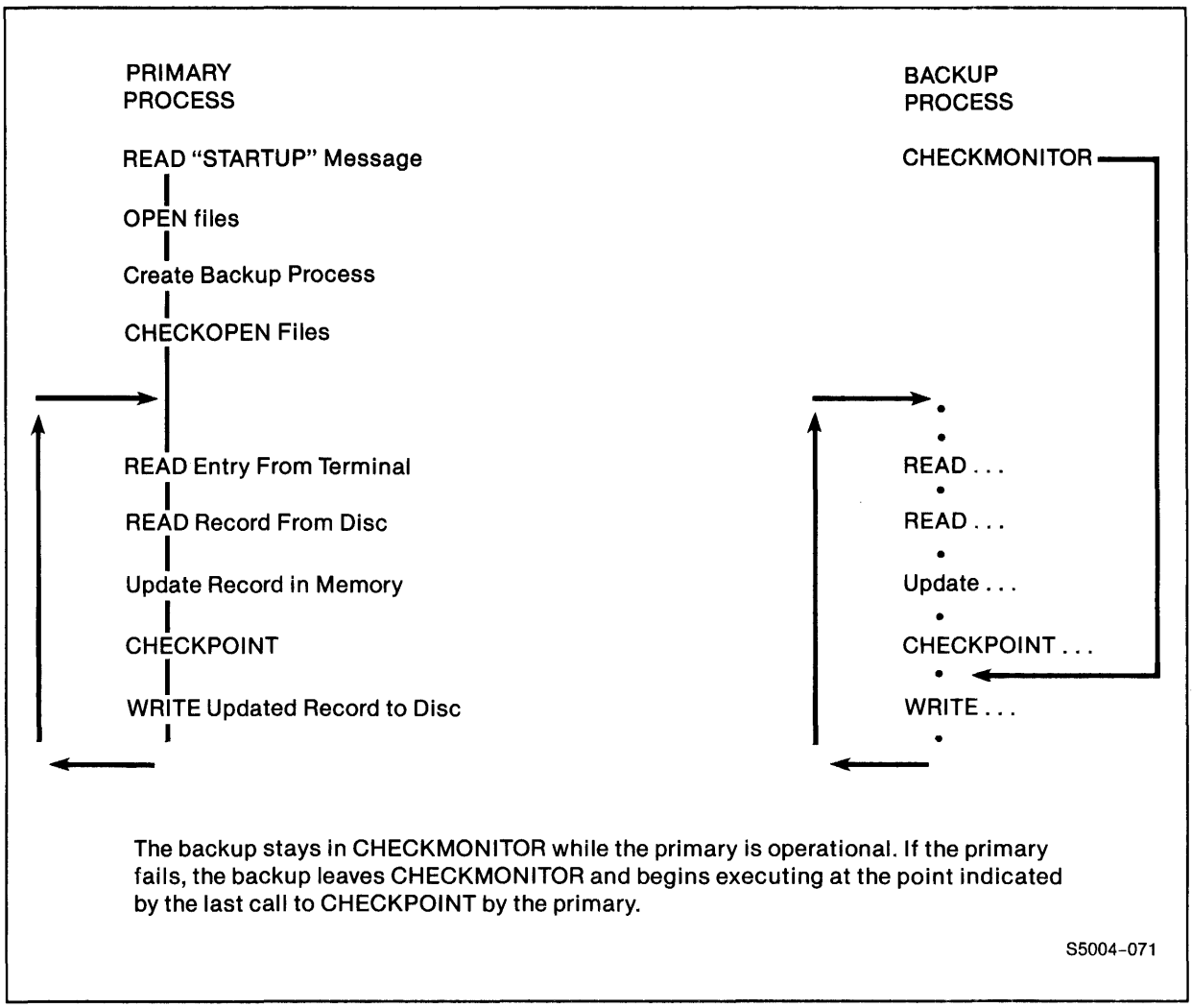


Figure 12-2. Fault-Tolerant Transaction Processing

Basically, the following actions take place when a program runs, as illustrated in Figure 12-2.

1. First, the program is given a process name at run time. This permits the new process (and eventually its backup) to run as a named process pair. (An alternate, though more complicated method of setting up a process pair is to use two nonnamed processes and have each call the STEPMOM procedure to "adopt" the other. This method is described in more detail later in this section.)

2. The new process determines that it is the primary process, and reads the startup message from its creator (for example, a command interpreter).

3. The primary process opens any files required for its execution.

4. The primary process then creates the backup process in another processor module. The backup process is given the same process name as the primary.

5. The backup process, at the beginning of its execution, determines that it is the backup process, and calls the CHECKMONITOR procedure. This is as far as the backup executes unless a failure of the primary process occurs.

6. The primary process opens the same files for the backup process by calls to CHECKOPEN. This permits files to be open by the pair in a manner that permits both members of the pair to have a file open while retaining the ability to exclude other processes from accessing a file. For disc files open in this manner, a record or file lock by the primary is also an equivalent lock by the backup.

7. The primary process then begins executing its main processing loop. At critical points through the execution loop, typically before writes to disc files, the primary calls CHECKPOINT to send part of its environment and pertinent file control information to the backup process. Typically, a program contains several calls to CHECKPOINT; each call checkpoints only a portion of the primary process's environment. Calls to CHECKPOINT that checkpoint the data stack define restart points for the backup process.

8. If the primary process fails, the backup begins executing at the restart point indicated by the latest call to CHECKPOINT that checkpointed the data stack. The backup process is now considered to be the primary process.

9. If the reason for the primary process failure was a processor module failure (CPU down), the new primary process creates a backup process when the failed processor module is repaired and brought back online. This new backup process is then ready to take over if the primary process fails. This is the normally recommended procedure; an alternative action is to create a backup process immediately in another CPU. A critical application usually requires an immediate backup.

The following summarizes the specific action of CHECKMONITOR for an action by the primary process. Refer to the System Procedure Calls Reference Manual for additional information.

Primary	Backup (CHECKMONITOR)
No action	At the beginning of CHECKMONITOR execution, the current state of CPU monitoring for the caller is saved (that is, the current MONITORCPUS <cpu-mask>), then MONITORCPUS is called, specifying only the primary's processor module.
CHECKOPEN	OPEN is called for the designated file.
CHECKPOINT	If all or a portion of the primary's data stack was checkpointed, the data is moved into the corresponding location in the backup's data stack. If a local data buffer was checkpointed by name, the data is moved into the appropriate location in the backup's data area. If file synchronization information was checkpointed, SETSYNCINFO is called for the designated file.
CHECKCLOSE	CLOSE is called for the designated file.
CHECKSWITCH	First CHECKMONITOR calls RESETSYNC for any file whose synchronization information was not checkpointed by the primary in its preceding call to CHECKPOINT. CPU monitoring is returned to the state that was in effect before CHECKMONITOR was called. Control is then returned to the point in the backup process indicated by the latest call to CHECKPOINT in the primary process. If the primary has not previously checkpointed its stack in a call to CHECKPOINT, control is returned to the instruction following the call to CHECKMONITOR.
Process Failure	(STOP or ABEND system message received for primary.) First CHECKMONITOR calls RESETSYNC for any file whose synchronization information was not checkpointed by the primary in its preceding call to CHECKPOINT. CPU monitoring is returned to the state that was in effect before CHECKMONITOR was called. Control is then returned to the point in the backup process indicated by the latest call to CHECKPOINT in the primary process. If the primary has not previously checkpointed its stack in a call to CHECKPOINT, control is returned to the instruction following the call to CHECKMONITOR.
Processor Failure	(Processor Failure system message received for primary's processor module.) First CHECKMONITOR calls RESETSYNC for any file whose synchronization information was not checkpointed by the primary in its preceding call to CHECKPOINT. CPU monitoring is returned to the state that was in effect before CHECKMONITOR was called. Control is returned to the point in the backup process indicated by the latest call to CHECKPOINT in the primary process. If the primary has not previously checkpointed its stack in a call to CHECKPOINT, control is returned to the instruction following the call to CHECKMONITOR.

FAULT-TOLERANT PROGRAM STRUCTURE

The general structure of a typical fault-tolerant program is:

- A main processing loop
- A process startup (beginning of program) phase

Please keep in mind that many of the examples in this section are incomplete; each segment illustrates only the point described. A complete example of a fault-tolerant program is given in Appendix B.

Main Processing Loop

In addition to normal transaction processing, the main processing loop for both a named and a nonnamed process pair must:

1. Checkpoint at appropriate points
2. Check the $RECEIVE file for system messages
3. Perform special action when taking over from the primary

Process Startup for Named Process Pairs

The use of named process pairs for fault-tolerant programming is considered to be the usual case. Nonnamed process pairs are used only in special cases.

The process startup code is executed by both the primary and backup processes following their creation.

In the following presentation, process startup for named process pairs is described before process startup for nonnamed process pairs. The same step sequence numbering is used for both. The following overview presents these individual steps concisely:

1. Save the stack base address for checkpointing.
2. Call ARMTRAP so the process will abend if trap occurs. (ARMTRAP can also be used to process traps; see Section 13.)
3. Determine if the process is the primary or the backup.

```
     If primary then:
4.     Open $RECEIVE (nowait) and, optionally, read the startup
         message
5.     Open files

6.     Monitor the backup CPU

7.     Create backup process:
         if created then
8.       open files in backup process

9.       use AWAITIO to complete nowait opens in backup process

10.      checkpoint environment to backup

11.  Else ! this is backup ! monitor the primary.

12.  Initiate a read on $RECEIVE to check for backup stopped, or
     processor up or processor down messages.
```

After performing these steps, execute the main processing loop.

1. SAVE THE STACK BASE ADDRESS: This is necessary for subsequent checkpointing of the data stack. The stack base address should be kept in a global variable. The stack base address is that of the first local variable of the main procedure:

```
INT .STACKBASE; ! global pointer variable.

PROC M MAIN;
  BEGIN
    INT .PPDENTRY [0:8],     ! first local variable in MAIN.
          .
          .
        BASE = 'L' + 1;  ! address equivalence.

    @STACKBASE := @BASE;  ! saves the address.
```

2. CALL ARMTRAP: The ARMTRAP procedure should be called to handle any trap that may occur. The simplest method of using ARMTRAP is

```
   CALL ARMTRAP ( 0, -1 );
```

This causes the process to abend if a trap occurs.

During the program debug phase, it is usually desirable to omit the call to ARMTRAP. Then, if a trap occurs, DEBUG (or INSPECT) is called. If you want the process to analyze the reason for the trap, see the information given in Section 13.

3. DETERMINE IF PRIMARY OR BACKUP: One way to determine if a process is a primary or its backup is to look at its entry in the process-pair directory (PPD):

```
INT .PPDENTRY [0:8];

CALL GETCRTPID ( MYPID, PPDENTRY );
CALL LOOKUPPROCESSNAME ( PPDENTRY );
IF < THEN CALL ABEND; ! no entry.
```

This returns the PPD entry for this process. If LOOKUPPROCESSNAME fails, either the process does not have a name or the system cannot access the PPD. In either case, a serious problem exists.

```
IF NOT PPDENTRY [4] THEN   ! i'm the primary
  BEGIN
    .
```

The fact that PPDENTRY [4] (<cpu2,pin2>) = 0 indicates that no backup process exists. Therefore, this process must be the primary.

The following actions are taken by the primary process:

4. OPEN $RECEIVE: The $RECEIVE file should be opened with nowait I/O specified. Nowait I/O is specified so that a read on $RECEIVE can be continually outstanding. This is desirable so that the "check for completion" form of AWAITIO (such as, <time-limit> = 0D) can be used to check for system messages or so that system messages can be read when waiting for completions on other files.

```
INT .RECEIVE[0:11] := ["$RECEIVE", 8 * [" "]],     ! global
     RFNUM,                                        ! variables
  .
  .
CALL OPEN ( RECEIVE, RFNUM, 1 );
IF < THEN CALL ABEND;
```

Next, if a startup message is expected (such as a command interpreter parameter message), it should be read:

```
CALL READ ( RFNUM, BUF, COUNT );
IF <> THEN CALL ABEND;
CALL AWAITIO ( RFNUM,, COUNTREAD );
IF <> THEN CALL ABEND;
```

At this point, a check should be made to determine if the message is a valid startup message (that is, if the first word of the message = -1).

5. OPEN PRIMARY'S FILES: The files to be referenced by the process should be opened by the primary (see "File Open").

```
LITERAL                                  ! global data
     FLAGS1       = ...,                 ! declarations.
     SYNC^DEPTH1  = ...,                 !
     FLAGS2       = ...,                 !
     SYNC^DEPTH2  = ...,                 !
        .                                !
        .                                !
     FLAGSN       = ...,                 !
     SYNC^DEPTHN  = ...;                 !
                                         !
INT .FNAME1 [0:11],                      !
     FNUM1,                              !
    .FNAME2 [0:11],                      !
     FNUM2,                              !
       .                                 !
       .                                 !
    .FNUMN [0:11],                       !
     FNUMN;                              !

CALL OPEN ( FNAME1, FNUM1, FLAGS1, SYNC^DEPTH1 );
IF < THEN .... ! see note.
CALL OPEN ( FNAME2, FNUM2, FLAGS2, SYNC^DEPTH2 );
IF < THEN .... ! see note.
  .
  .
CALL OPEN ( FNAMEN, FNUMN, FLAGSN, SYNC^DEPTHN );
IF < THEN .... ! see note.
```

NOTE

The action that should be taken if a file open fails (IF ERROR <> 0) is application-dependent. For example, the primary could abort itself. Or, if an invalid file name was received by the process, the terminal operator could be queried for a valid file name.

6. MONITOR THE BACKUP CPU: The MONITORCPUS procedure should be called for the backup process's processor module. This allows the processor module failure and reload system messages to be sent to the primary process (through the $RECEIVE file).

```
INT BACKUP^CPU; ! backup CPU no.

  .
  ! monitor the backup CPU.
CALL MONITORCPUS ( %100000 '>>' BACKUP^CPU );
  .
```

7. CREATE THE BACKUP PROCESS: Backup process creation is best accomplished by writing a procedure that performs the following functions:

- Creating the process
- Opening the files on behalf of the backup
- Checkpointing the primary's environment

The reason for including these functions in a procedure is that backup process creation may be necessary at several points during process execution. These are: during process startup, after a takeover by a backup following a failure of its primary, failure of backup (ABEND), or reload of the backup's processor module.

The following is an example of backup process creation:

```
PROC CREATEBACKUP;
   BEGIN
```

Create the process:

```
INT .PFILE [0:11],
     PNAME [0:3],
     BACKUP^PID [0:3],
     ERROR;

CALL PROGRAMFILENAME ( PFILE );
```

Returns the file name of the primary's program file.

```
CALL GETCRTPID ( MYPID, PNAME );
```

Returns the process pair's name.

```
CALL NEWPROCESS ( PFILE,, (LASTADDR'>>'10) '+' 1,
                  BACKUP^CPU, BACKUP^PID, ERROR, PNAME );
```

Creates the process. (For an explanation of the LASTADDR procedure, see the System Procedure Calls Reference Manual.)

Open the files in the backup process (see "File Open" for considerations):

```
IF BACKUP^PID THEN ! it was created.
  BEGIN

    BACKUP^UP := 1; ! global variable.

    ! $RECEIVE file.
    CALL CHECKOPEN ( RECEIVE, RFNUM, 1,,,, ERROR );
```

```
IF <> THEN ... ! see note.
CALL CHECKOPEN(FNAME1,FNUM1,FLAGS1,SYNC^DEPTH1,,,ERROR);
IF <> THEN ... ! see note.
CALL CHECKOPEN(FNAME2,FNUM2,FLAGS2,SYNC^DEPTH2,,,ERROR);
IF <> THEN ... ! see note.
  .
  .
CALL CHECKOPEN(FNAMEN,FNUMN,FLAGSN,SYNC^DEPTHN,,,ERROR);
IF <> THEN ... ! see following paragraphs.
```

NOTE

The action that a primary should take if a file open in its backup fails (if error <> 0) is application-dependent. For example, the primary could stop the backup, then abort itself. Or, the primary could stop the backup but continue processing without a backup. If the latter course of action is taken, however, the primary will receive a process STOP system message for the backup. Therefore, the primary should contain logic so that it does not recreate its backup if this happens.

When a server is opened in a nowait manner by a process pair, the OPEN and the CHECKOPEN must both have been completed without error by AWAITIO before the sync block is checkpointed. If this restriction is not obeyed, the CHECKOPEN is rejected with an error, and a takeover occurs, then the server may not recognize the backup as a valid opener. In this case, pending requests may be rejected with an error if retried without the backup process first opening the file on its own. When using nowait opens, the primary process of a pair should create the backup in the following manner to ensure a valid takeover:

1. Create backup using the NEWPROCESS procedure.

2. CHECKOPEN all files.

3. Complete all nowait CHECKOPENs by calls to AWAITIO.

4. Checkpoint the stack and sync blocks.

If the primary process dies, the backup is now ready to continue processing. Normal processing can continue in parallel with step 3, which may take a while if one or more servers responds slowly.

Checkpoint the primary's data area to the backup process (this includes any startup message):

```
   CALL CHECKPOINT (, ADDR, COUNT, ... );
            .
```

Checkpoint all files' sync information and the data stack in the same call:

```
            .
            .
   ! set restart point.
   IF (STATUS := CHECKPOINT(STACKBASE,, FNUM1,, FNUM2,, ...
                            ,, FNUMN )) THEN

      CALL ANALYZE^CHECKPOINT^STATUS ( STATUS );

      ANALYZE^CHECKPOINT^STATUS is a procedure that takes
      appropriate action for a checkpoint failure or takeover by
      backup.  See "Takeover by Backup" for a description of the
      ANALYZE^CHECKPOINT^STATUS procedure.
```

If multiple calls to CHECKPOINT are necessary, the data stack should be checkpointed last. This checkpoint is then a restart point if the primary should subsequently fail.

```
          END; ! open files
     END; ! of createbackup
```

11. MONITOR THE PRIMARY: This is the action taken by the process if it is the backup. First, MONITORCPUS is called for the primary's processor module (this is done so that the primary's processor module will continue to be monitored if and when the backup takes over). The actual monitoring of the primary is accomplished by calling the CHECKMONITOR procedure:

```
   ! save the primary's CPU number.
   BACKUP^CPU := PPDENTRY [3].<0:7>;
   ! monitor the primary CPU.
   CALL MONITORCPUS ( %100000 '>>' BACKUP^CPU );
   CALL CHECKMONITOR;
   CALL ABEND;
```

The backup process only returns from the call to CHECKMONITOR if the primary has not checkpointed its data stack. The primary checkpoints its stack for the first time at the end of creation of the backup process.

12. READ $RECEIVE: The primary should keep a read outstanding on $RECEIVE at all times. This is desirable so that process deletion, and processor failure and reload system messages can be received.

```
    CALL READ ( RFNUM, RBUF, COUNT );
```

EXAMPLE: The following code is an example of process startup for named process pairs:

```
INT  BACKUP^CPU,                                    ! global data
    .STACKBASE, ! global pointer variable.          ! declarations.
    .RECEIVE[0:11] := ["$RECEIVE", 8 * ["  "]],     !
     RFNUM,                                         !
     STOP^COUNT := 0,                               !
     BACKUP^UP := 0;                                !

LITERAL                                             !
     FLAGS1      = ...,                             !
     SYNC^DEPTH1 = ...,                             !
     FLAGS2      = ...,                             !
     SYNC^DEPTH2 = ...,                             !
        .                                           !
        .                                           !
     FLAGSN      = ...,                             !
     SYNC^DEPTHN = ...;                             !
                                                    !
INT .FNAME1 [0:11],                                 !
     FNUM1,                                         !
    .FNAME2 [0:11],                                 !
     FNUM2,                                         !
       .                                            !
       .                                            !
    .FNUMN[0:11],                                   !
     FNUMN;                                         !

  .
PROC M MAIN;
  BEGIN
    INT .PPDENTRY [0:8],   ! first local variable in MAIN.
          .
          .
        BASE = 'L' + 1;  ! address equivalence.

    @STACKBASE := @BASE;  ! save the address.

      ! abort the process if a trap occurs.
    CALL ARMTRAP ( 0, -1 );

    CALL GETCRTPID ( MYPID, PPDENTRY);
    CALL LOOKUPPROCESSNAME ( PPDENTRY );
    IF < THEN CALL ABEND; ! no entry.

    IF NOT PPDENTRY [4] THEN ! i'm the primary
      BEGIN
          ! OPEN $RECEIVE.
        CALL OPEN ( RECEIVE, RFNUM, 1 );
        IF < THEN CALL ABEND;
          ! read the startup message.
```

```
        CALL READ ( RFNUM, BUF, COUNT );
        IF <> THEN CALL ABEND;
        CALL AWAITIO ( RFNUM,, COUNTREAD );
        IF <> THEN CALL ABEND;

          ! open the primary's files.
        CALL OPEN ( FNAME1, FNUM1, FLAGS1, SYNC^DEPTH1 );
        IF < THEN .... ! error.
        CALL OPEN ( FNAME2, FNUM2, FLAGS2, SYNC^DEPTH2 );
        IF < THEN .... ! error.
          .
          .
        CALL OPEN ( FNAMEN, FNUMN, FLAGSN, SYNC^DEPTHN );
        IF < THEN .... ! error.

          ! monitor the backup CPU.
        CALL MONITORCPUS ( %100000 '>>' BACKUP^CPU );

          ! create the backup process.
        CALL CREATEBACKUP ( BACKUP^CPU );
      END
    ELSE ! i'm the backup
      BEGIN
          ! save the primary's CPU num.
        BACKUP^CPU := PPDENTRY [3].<0:7>;
          ! monitor the primary CPU.
        CALL MONITORCPUS ( %100000 '>>' BACKUP^CPU );
        CALL CHECKMONITOR;
        CALL ABEND;
      END;

    ! read $RECEIVE.
    CALL READ ( RFNUM, RBUF, COUNT );

    ! execute the main program loop.
    CALL MAIN^LOOP;
  END;
```

The following is the example code in the CREATEBACKUP procedure:

```
PROC CREATEBACKUP (BACKUP^CPU ;
   INT BACKUP^CPU;

  BEGIN
    INT .PFILE [0:11],
        PNAME [0:3],
        BACKUP^PID [0:3],
        ERROR;

    CALL PROGRAMFILENAME ( PFILE );

    CALL GETCRTPID ( MYPID, PNAME );
```

```
    CALL NEWPROCESS (PFILE, , (LASTADDR'>>'10)'+'1, BACKUP^CPU,
    BACKUP^PID, ERROR, PNAME);

    IF BACKUP^PID THEN ! it was created.
      BEGIN
        BACKUP^UP := 1;
        CALL CHECKOPEN(RECEIVE,RFNUM,1,,,,ERROR) ! $RECEIVE file.
        IF <> THEN ... ! error.
        CALL CHECKOPEN(FNAME1,FNUM1,FLAGS1,SYNC^DEPTH1,,,ERROR);
        IF <> THEN ... ! error.
        CALL CHECKOPEN(FNAME2,FNUM2,FLAGS2,SYNC^DEPTH2,,,ERROR);
        IF <> THEN ... ! error.
          .
          .
        CALL CHECKOPEN(FNAMEN,FNUMN,FLAGSN,SYNC^DEPTHN,,,ERROR);
        IF <> THEN ... ! error.

        CALL CHECKPOINT (,, FNUM1,, FNUM2,, .......,, FNUMN );
        CALL CHECKPOINT (, ADDR, COUNT, ... );
          .
          .
        IF (STATUS := CHECKPOINT(STACKBASE)) THEN ! restart point
           CALL ANALYZE^CHECKPOINT^STATUS ( STATUS );
      END; ! open files
  END; ! of createbackup
```

Process Startup for Nonnamed Process Pairs

The startup for nonnamed process pairs is nearly identical to that for named process pairs, except for the following items:

- The determination of primary or backup designation is not based on a PPD entry.
- The primary must send a startup message to the backup.
- The backup must call the STEPMOM procedure for the primary. This is necessary because the checkpointing facility uses the creator process ID in the primary's process control block (PCB) to determine the destination of checkpoint messages.
- The startup message must be read by using the READUPDATE procedure (and, therefore, replied to by using the REPLY procedure). This is done so that the primary process is suspended (and therefore prevented from checkpointing) until the backup calls the STEPMOM procedure.

NOTE

There is no "ancestor" relationship between a nonnamed process pair and the process initially responsible for their creation.

In the following list of the general steps involved in process startup for nonnamed processes, the differences from the startup process for named process pairs are indicated by lettered steps, described in detail below:

1. Save the stack base address for checkpointing.

2. Call ARMTRAP, so process will abend if trap occurs. (ARMTRAP can also be used to process traps, see Section 13.)

A. Open $RECEIVE (nowait, receive depth = 1) and, read the startup message by using READUPDATE.

B. Determine if the process is the primary or backup -

```
     If primary then
       begin
C.       reply to startup message

5.       Open files

6.       Monitor the backup CPU

7.       Create backup process:
           if created then
             begin
D.             send nonstandard startup message to backup

8.             open files in backup process

9.             checkpoint environment to backup
             end
       end
E.   else ! backup ! monitor the primary.
```

12. Initiate a read on $RECEIVE to check for backup stopped, or processor up or down messages.

After performing these steps, execute the main program loop.

A. READ STARTUP MESSAGE: The $RECEIVE file should be opened with nowait I/O and <receive-depth> >= 1 specified. <receive-depth> >= 1 is specified so that the startup message can be read by a call to READUPDATE, then later replied to by a call to REPLY. This is necessary so that the backup process, after it

reads its startup message, can cause the primary process to be suspended until it has a chance to call the STEPMOM procedure on the primary process.

```
INT .RECEIVE [0:11] := ["$RECEIVE", 8 * ["  "]], ! global
     RFNUM,                                      ! variables.

  .
CALL OPEN ( RECEIVE, RFNUM, 1, 1 );
IF < THEN CALL ABEND;
```

Next, the startup message (such as the command interpreter param message) is read:

```
CALL READUPDATE ( RFNUM, BUF, COUNT );
IF <> THEN CALL ABEND;
CALL AWAITIO ( RFNUM,, COUNTREAD );
IF <> THEN CALL ABEND;
```

The call to READUPDATE causes the sender of the startup message to be suspended until the message is replied to. At this point, a check should be made to determine if the message is a valid startup message (if the first word of the message = -1).

B. DETERMINE IF PRIMARY OR BACKUP: A recommended way to designate whether a nonnamed process is a primary or its backup, is to have the primary process send a nonstandard startup message to the backup after the backup's creation. Then, if the new process reads a standard startup message, it knows that it is the primary; otherwise, it knows that it is the backup. A recommended form for a nonstandard startup message is:

```
<startup-message> [0] = -1
<startup-message> [1] = -2
```

The first word of the startup is the same as the command interpreter's startup message (this allows the program logic for checking for a valid startup message to be the same for both the primary and the backup). The designation of primary or backup is made by checking word[1] of the startup message:

```
IF BUF [1] <> -2 THEN ! i'm the primary
   BEGIN
     .
```

A startup message from the command interpreter contains the "default volume/subvolume" names starting in word [1]. Therefore, word [1].<0:7> = "$" for a standard command interpreter startup message.

C. REPLY TO STARTUP MESSAGE: The primary process must reply to the startup message by a call to the REPLY procedure:

```
CALL REPLY;
```

permits the command interpreter to continue executing.

D. SEND NONSTANDARD STARTUP MESSAGE TO BACKUP: A nonstandard startup message is sent to the backup following the backup's creation. The nonstandard startup message provides the primary or secondary designation for the process pair:

```
IF BACKUP^PID THEN ! it was created.
  BEGIN

    ! open a file to the backup process.
    CALL OPEN ( BACKUP^PID, FNUM );
    IF <> THEN ... ! couldn't open backup.  Bad news.

    ! build nonstandard startup message.
    BUF [0] := -1;
    BUF [1] := -2;

    ! send the startup message.
    CALL WRITE ( FNUM, BUF, 4 );
    IF <> THEN ... ! couldn't write to backup.  Bad news.

    The primary process is suspended at this point until
    the backup process replies to the startup message.

    ! close the file to the backup process
    CALL CLOSE ( FNUM );

    BACKUP^UP := 1;

    ! open files for backup process.
    CALL CHECKOPEN(FNAME1,FNUM1,FLAGS1,SYNC^DEPTH1,,,ERROR);
    IF <> THEN ... ! error.
      .
```

E. MONITOR THE PRIMARY: This is the action taken by the process if it is the backup. First, MOM is called to get the process ID of the primary process. Second, STEPMOM is called for the primary process (this is necessary so that the backup process will receive the checkpoint messages sent when the primary calls CHECKPOINT). Next, REPLY is called to reply to the startup message (this allows the primary to resume execution and make its first call to CHECKPOINT). Then, MONITORCPUS is called for the primary's processor module (this is done so that the primary's processor module will continue to be monitored if and when the backup takes over). The actual monitoring of the primary is accomplished by calling the CHECKMONITOR procedure:

```
CALL MOM ( BACKUP^PID );
CALL STEPMOM ( BACKUP^PID );
IF < THEN CALL ABEND;
CALL REPLY;
  ! save the primary's CPU number.
BACKUP^CPU := BACKUP^PID [3].<0:7>;
  ! monitor the primary CPU.
CALL MONITORCPUS ( %100000 '>>' BACKUP^CPU );
CALL CHECKMONITOR;
CALL ABEND;
```

The backup process only returns from the call to CHECKMONITOR if the primary has not checkpointed its data stack. The primary checkpoints its stack for the first time at the end of creation of the backup process.

FILE OPEN

Files are opened in a primary process by calls to the OPEN procedure.

For disc files, when automatic path error recovery is desired, the number of write operations whose outcome the system is to remember is specified in the <sync-depth> parameter to OPEN.

Files are opened for a backup process by its primary process through calls to the CHECKOPEN procedure.

The use of CHECKOPEN permits both members of a process pair to have a file open, while retaining the ability to exclude other processes from accessing a file. For disc files open in this manner, a record or file lock by the primary is also an equivalent lock by the backup.

Note that the same parameter values that are passed to OPEN are also passed to CHECKOPEN; both files must be open with the same <filenum>, <flags> value, and <sync-depth> value; for example, in the primary process:

```
LITERAL FLAGS1 = ...,
      SYNC^DEPTH1 = ...;

INT .FNAME1 [0:11],
      FNUM1,
      ERROR;

      .
  ! open the file for the primary.
CALL OPEN ( FNAME1, FNUM1, FLAGS1, SYNC^DEPTH1 );
```

```
IF <> THEN ... ! error occurred.
     .
  ! open the file for the backup.
CALL CHECKOPEN (FNAME1, FNUM1, FLAGS1, SYNC^DEPTH1,,, ERROR);
IF <> THEN ... ! error occurred.
```

CHECKPOINTING

Checkpoints are used to preserve transaction data and to identify a restart point in the event of a failure. For each checkpoint in a primary process, there is a corresponding restart point in its backup process, as shown in Figure 12-3.

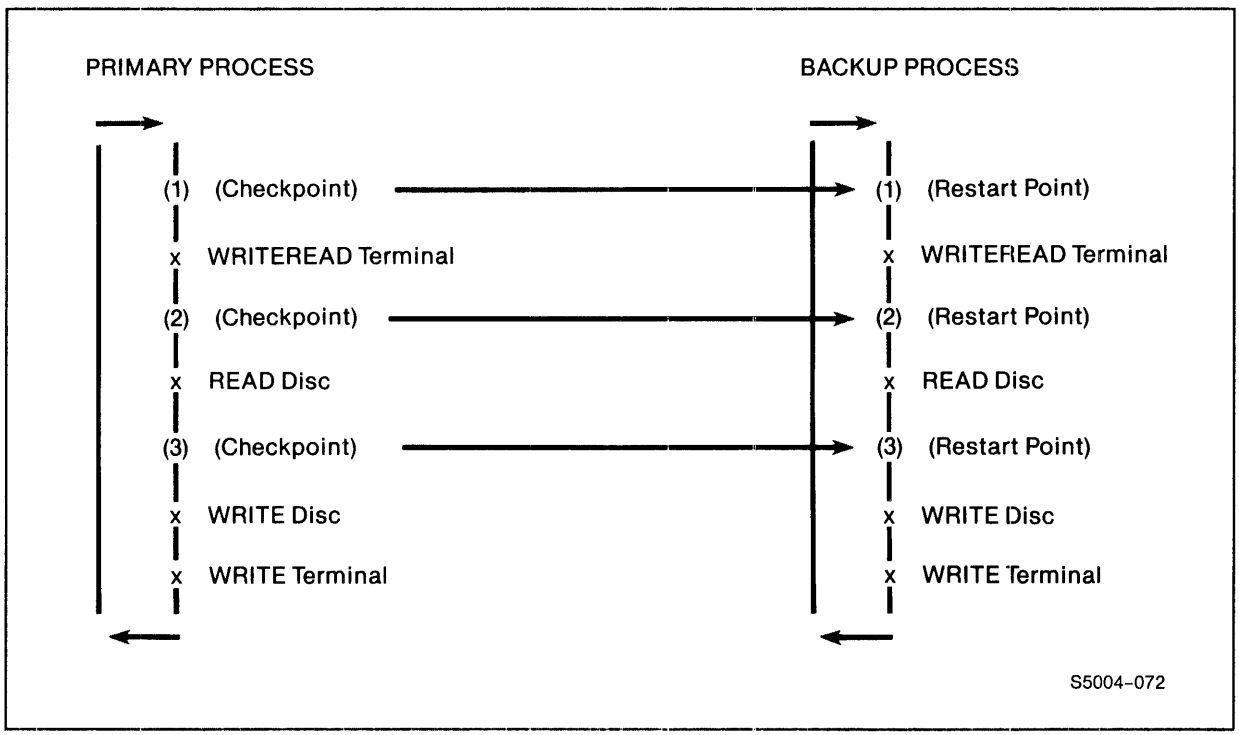


Figure 12-3. Checkpoints and Restart Points

Enough checkpoints must be provided, and each must contain enough information, so that in the event of a failure in the primary process, the backup process can take over the process pair's duties while maintaining the integrity of any data involved in the current transaction.

The amount of checkpointing that you must provide depends on the degree of recoverability you must have. As an extreme example, a

primary process could, after execution of each program statement, send its entire data area and its current program counter setting. A program of this type would be recoverable after each statement. Because of the amount of system resources needed, this type of checkpointing would be extremely time consuming and inefficient.

Processes typically checkpoint only elements that have changed since the last call to CHECKPOINT was made. This minimizes the checkpoint message length and message-handling overhead.

From a practical standpoint, checkpointing internal calculations is not necessary, as they can be performed with virtually no loss of system throughput. Checkpointing is necessary only when data is being transferred between the internal program environment and a file. For example, the primary process may checkpoint the data just read from a terminal so that if a failure occurs, the terminal operator will not have to reenter the data.

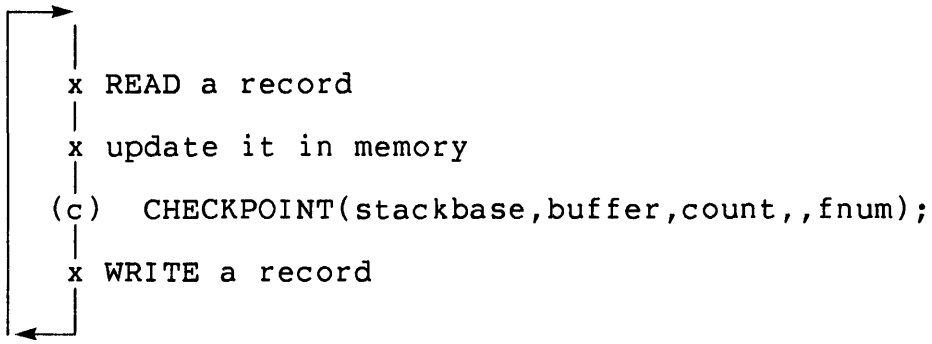
Guidelines for Checkpointing

As a general rule, a call to CHECKPOINT should immediately precede:

- Any write to a file (including a WRITEREAD to a terminal)
- A call to CONTROL or SETMODE for a file

To provide a greater degree of recoverability, a call to CHECKPOINT may immediately follow:

- A read from a terminal



The call to CHECKPOINT should checkpoint the following:

- A value or set of values indicating the program state. This is usually accomplished by checkpointing the process's data stack.
- If the checkpoint precedes a write to disc file, the file's sync block.
- The file's data buffer. If the data buffer is within the memory stack area (actually, from the application-defined stack base through the address indicated by the current S register setting), the data buffer will be checkpointed when the stack is checkpointed.

Adherence to the above guidelines will assure that an application program can recover from disc file operations and, in most cases, terminal operations.

You should strive to keep to a minimum the number of checkpoints in a processing loop and the amount of data checkpointed in a given call to CHECKPOINT. One approach is to checkpoint only a portion of the program state (for example, some data buffers and/or the data stack) at one time. You must take care that any checkpoint which is also a restart point (includes the data stack) yields a valid program state.

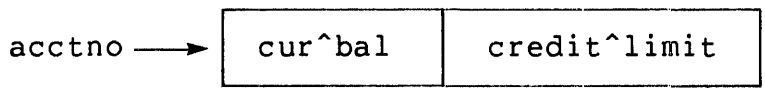
You must also take care when checkpointing a data buffer without checkpointing the data stack, that the preceding restart point is still valid (does not use the new value of the data buffer). Proper checkpointing can be achieved only by careful analysis of the operation being performed and of the intended checkpoints and their contents.

Note that I/O to nondisc and nonterminal devices involves very application-dependent recovery procedures. For example, a report to a line printer may have to be restarted from the last page, or a magnetic tape may have to be repositioned.

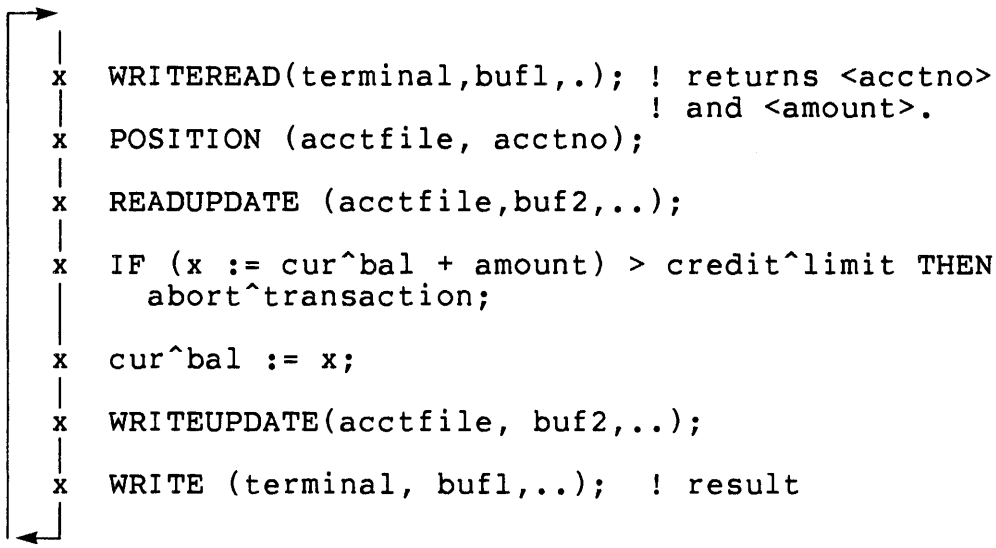
Example of Where Checkpoints Should Occur

The following is an example of a simple transaction cycle to update a record.

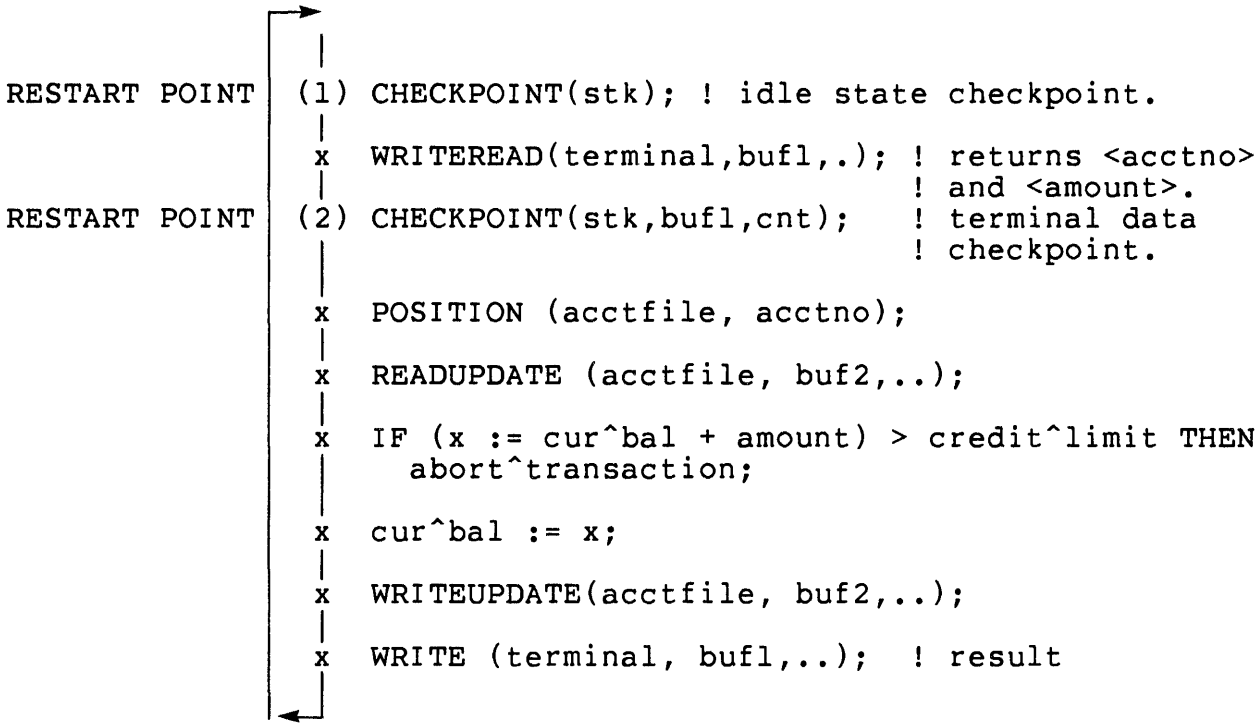
The record:



The transaction cycle (without checkpoints):



The transaction cycle with insufficient checkpoints:



In this last example, the first checkpoint identifies the program state as being idle (or waiting from input from the terminal). The actual checkpoint message consists only of the primary process's data stack.

The second checkpoint identifies the program state as "terminal entry just read". The checkpoint message consists of two parts:

1. The primary's data stack

2. The data read from the terminal

Here the assumption is that, because the transaction is driven by the data read from the terminal, this data is ample for the backup to perform the identical operation. This assumption is incorrect, however. A problem occurs if a failure occurs just following the WRITEUPDATE of the "acctfile". This is illustrated in the following transaction:

```
WRITEREAD(terminal, buf1,..); returns: "acctno" = "12345",
                                       "amount" = "$10"

(2) checkpoint "12345, $10"

POSITION (acctfile, 12345D);

READ (acctfile,buf2,..);
```

returns:	"acctno"	"cur^bal"	"credit^limit"
	12345 →	$485	$500

```
IF (x := $485 + $10) > $500 THEN ...

cur^bal := x;

WRITEUPDATE (acctfile,buf2,..);
```

writes:	"acctno"	"cur^bal"	"credit^limit"
	12345 →	$495	$500

```
****** FAILURE HERE ******
```

Backup's restart with latest checkpoint data: "12345, $10"

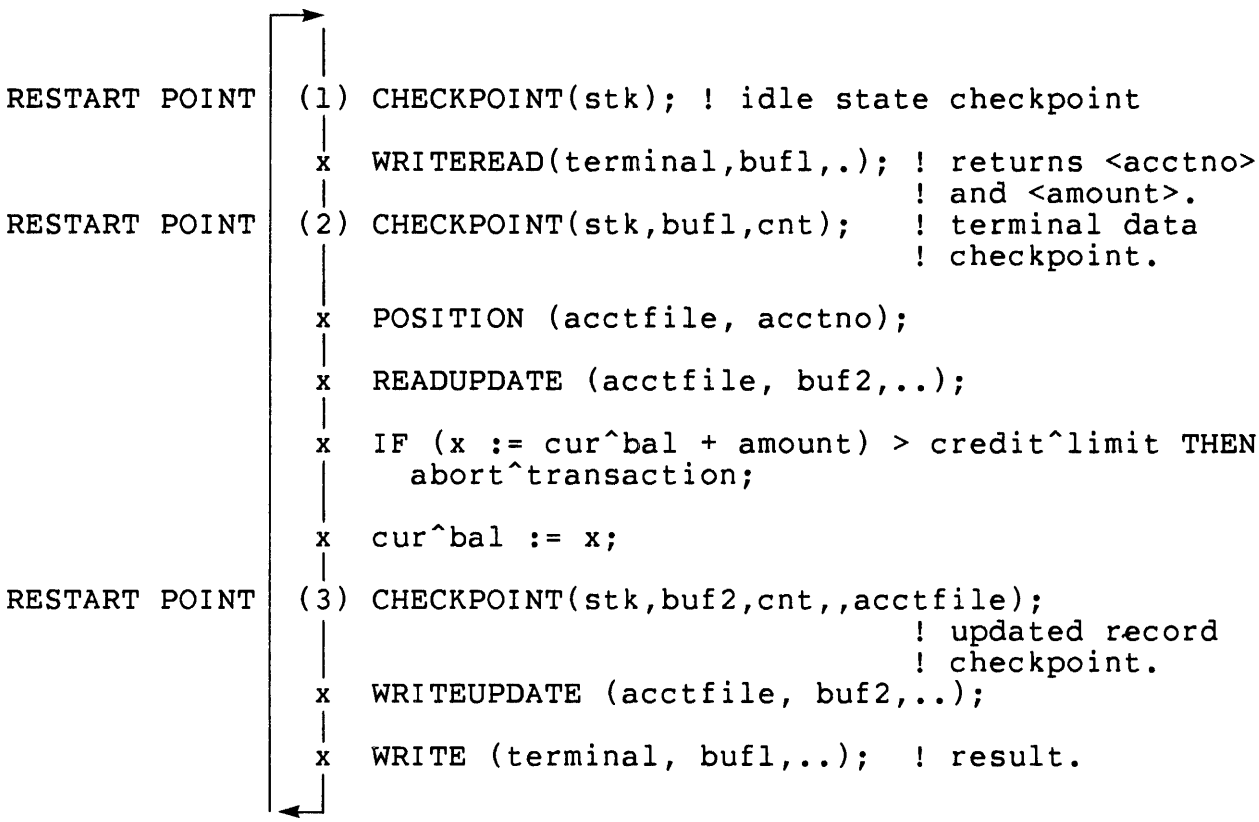
```
POSITION (acctfile, 12345D);

READ (acctfile,buf2,..);
```

returns:	"acctno"	"cur^bal"	"credit^limit"
	12345 →	$495	$500

```
IF (x := $495 + $10) > $500 THEN ...
```

Here the test fails because the update to the disc completed successfully and the "cur^bal" has already been updated. The terminal operator is given an indication that "acctno" 12345 has attempted to exceed its credit limit; therefore, the purchase is refused. However, account 12345's balance reflects that a purchase was made.

The transaction cycle with sufficient checkpointing:



The additional third checkpoint, (3), identifies the program state as "preparing to write an updated disc record to the disc". The checkpoint consists of three parts:

1. The primary process's stack

2. The disc file's sync information

3. The updated record

If the primary process fails between checkpoints 1 and 2, the backup process reissues the WRITEREAD to the terminal. If the primary process fails between checkpoints 2 and 3, the backup uses the terminal entry and continues the processing of the transaction. If the primary process fails subsequent to

checkpoint 3, the backup uses the latest checkpointing information to reexecute the write to disc.

Note that checkpoint (2) and its associated restart point could be omitted. If this were done, a failure between checkpoints (2) and (3) would require the operator to reenter the transaction.

Checkpointing Multiple Disc Updates

When performing a series of updates to one or more disc files, the checkpoint for those updates can be performed at one point in the program. The result is less system usage than that required for several checkpoints.

The program should be structured so that the series of writes needed to update a file are performed in a group. For each file to be checkpointed in this manner, the <sync-depth> parameter value of OPEN is specified as the maximum number of calls to WRITE for the file that are made between checkpoints for the file. Then, when a file is about to be updated by performing <sync-depth> writes to the file, the file's sync block and the data buffers about to be written to the file are checkpointed. In any case, care must be taken to ensure the integrity of any data referenced.

Considerations for Nowait I/O

When taking over from a failure of the primary, any nowait operations initiated, but not completed, by the primary before its failure, must be reinitiated by the backup; for example:

```
CALL READ ( RFNUM, RBUFFER, COUNT ); ! nowait.
   .
   .
CALL WRITE ( FNUM1, BUFFER, COUNT );    ! nowait.

CALL CHECKPOINT ( STACKBASE );

FNUM := -1;
CALL AWAITIO ( FNUM, .. );              ! wait on any completion.
```

If a failure occurs, the backup begins executing following the call to CHECKPOINT. However, there will be no outstanding I/O operations.

A solution might be to checkpoint before the I/O operations are initiated. However, in the case of $RECEIVE, because the process could need have a read continually outstanding, this may not be possible. For $RECEIVE, the read can be reinitiated when the backup takes over.

Action for CHECKPOINT Failure

If an UNABLE TO COMMUNICATE WITH BACKUP error occurs when checkpointing (CHECKPOINT.<0:7> = 1 on return), the primary process should stop the backup process. The primary process should then create a new backup process when the STOP system message (system message -5) is received. If the checkpoint failure persists, the failure should be noted accordingly, and the primary should stop the creation attempts. (See the ANALYZE^CHECKPOINT^STATUS example procedure under the heading "Takeover by Backup" in this section.)

NOTE

A checkpoint failure of this type normally indicates a system resource problem caused by application process checkpoints that are too large.

SYSTEM MESSAGES

The following system messages are related to recovery from process and processor module failures. Their formats, in word elements, are

- CPU Down Message. There are two forms of 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.

and

```
<sysmsg>            = -2
<sysmsg>[1] FOR 3   = $<process-name>
<sysmsg>[4]         = -1
```

This form is received by an ancestor process when the indicated process name is deleted from the PPD because of a processor module failure. This means that the named process or process pair no longer exists.

NOTE

Following a takeover by a backup process because of a processor module failure, the backup process, if it is an ancestor process, can expect to receive the second form of the CPU Down message. This message is received when a descendant process or process pair of the backup no longer exists because of the failure. One of these messages is received for each descendant process or process pair of the backup that disappears because of the processor module failure.

- CPU Up Message

```
<sysmsg>            = -3
<sysmsg>[1]         = <cpu>
```

This message is received if a reload occurs with a processor module being monitored.

- Process Normal Deletion (STOP) Message

This message is received if a process deletion is due to a call to the process control STOP procedure.

There are two forms of the STOP message:

```
<sysmsg>            = -5
<sysmsg>[1] FOR 4   = process ID of deleted process,
```

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>            = -5
<sysmsg>[1] FOR 3   = $<process-name> of deleted process
<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.

- Process Abnormal Deletion (ABEND) Message

  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.

  There are two forms of the ABEND message:

  ```
  <sysmsg>              = -6
  <sysmsg>[1] FOR 4     = process ID of deleted process
  ```

  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
  <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.

Recommended Action

The following is the recommended action when the above messages are received:

Message Number	Action by Primary
-2, CPU down:	Ignore it. An exception to this is if the second form of the CPU down message is received; the "ancestor" process may desire to recreate the failed process or process pair.
-3, CPU up:	Create the backup, and so on.
-5, backup stopped:	This shouldn't happen, but if it does, create the backup.
-6, backup abended:	Create the backup, etc.
other messages:	Take application-dependent action.

For system messages -5 and -6, the program should assure that the primary process does not loop continuously because of continually failing backup process.

Following a read of a system message, a read on the $RECEIVE file should be initiated.

The following is an example procedure that analyzes system messages and takes appropriate action:

```
PROC ANALYZE^SYSTEM^MESSAGE;

  BEGIN
    CASE $ABS ( RBUF ) OF      ! First word of $RECEIVE buffer
      BEGIN
        ;      ! 0.
        ;      ! 1.
        BEGIN ! 2 = CPU down.
          BACKUP^UP := 0;
        END;
        BEGIN ! 3  = CPU up.
          STOP^COUNT := 0; ! this must be checkpointed.
          CALL CREATE^BACKUP ( BACKUP^CPU );
        END;  ! 3.
        ;      ! 4.
        BEGIN ! 5 = backup stopped.
          BACKUP^UP := 0;
          STOP^COUNT := STOP^COUNT + 1;
          CALL CREATE^BACKUP ( BACKUP^CPU );
        END;  ! 5.
        BEGIN ! 6 = backup abended.
          BACKUP^UP := 0;
          STOP^COUNT := STOP^COUNT + 1;
          CALL CREATE^BACKUP ( BACKUP^CPU );
        END;  ! 6.
        OTHERWISE ! other system message.
          BEGIN
            .
            .
            .
          END;
      END; ! case of system message.
    ! issue a read to $RECEIVE.
    CALL READ ( RFNUM, RBUF, COUNT );
  END; ! analyze^system^message.
```

The STOP^COUNT variable is used to detect repeated backup process failures that are not due to processor module failures. This variable is cleared when a CPU Up message is received. Fault-tolerant programs should include such a variable to ensure that the primary process does not loop, continually recreating its backup. If STOP^COUNT reaches a count of 10, then the problem should be noted (a console message should be logged), and no further attempt at creation should occur until the problem is corrected.

TAKEOVER BY BACKUP

The following is the recommended action by the backup when it takes over from the primary. The action taken is dependent on the reason for the takeover:

- If return is from CHECKMONITOR, call ABEND (primary's stack has not been checkpointed).
- If return is from CHECKPOINT, then:

Reason (CHECKPOINT.<8:15>)	Action
0, primary stopped	Call STOP.
1, primary abended	Create backup, open its files, etc.
2, primary CPU down	None (this will be taken care of when a subsequent CPU Up system message is received).
3, primary called CHECKSWITCH	None.
Any except 0	Issue a read on $RECEIVE.

The example procedure on the next page analyzes the value returned from CHECKPOINT and takes appropriate action.

```
PROC ANALYZE^CHECKPOINT^STATUS ( STATUS );
   INT STATUS; ! return value of CHECKPOINT.

  BEGIN
    INT .BACKUP^PID [0:3];

    IF BACKUP^UP THEN ! analyze it.
      CASE STATUS.<0:7> OF
        BEGIN
          ;      ! 0 = good checkpoint.
          BEGIN ! 1 = checkpoint failure.
            ! find out if backup is still running.
            CALL MOM ( BACKUP^PID );
            CALL GETCRTPID ( BACKUP^PID [3], BACKUP^PID );
            IF = THEN ! backup still running.
              BEGIN
                ! stop the backup.
                CALL STOP ( BACKUP^PID );
                BACKUP^UP := 0;
              END;
          END;   ! 1.
          BEGIN ! 2 = takeover from primary.
            CASE STATUS.<8:15> OF
              BEGIN
                ! 0 = primary stopped.
                CALL STOP;
                ! 1 = primary abended.
                BEGIN
                  BACKUP^UP := 0;
                  STOP^COUNT := STOP^COUNT + 1;
                  CALL CREATE^BACKUP ( BACKUP^CPU );
                END;
                ! 2 = CPU down.
                BACKUP^UP := 0;
                ! 3 = primary called CHECKSWITCH.
                ;
              END; ! case of STATUS.<8:15>.
            ! issue a read to $RECEIVE.
            CALL READ ( RFNUM, RBUF, COUNT );
          END;   ! 2.
          BEGIN ! 3 = bad parameter to CHECKPOINT
            CALL DEBUG;  ! for testing checkpoint code.
          END;   ! 3.
        END; ! case of STATUS.<0:7>.
  END; ! ANALYZE^CHECKPOINT^STATUS.
```

See the ANALYZE^SYSTEM^MESSAGE procedure on the previous page for an explanation of the STOP^COUNT variable.

OPENING A FILE DURING PROCESSING

When files are opened after process startup, the possibility exists that a failure could occur during the file open. This can result in the backup process opening the same file twice. The following is a recommended procedure for opening a file during processing:

```
INT PROC FILEOPEN (FILENAME, FNUM, FLAGS, SYNCDEPTH);
 INT .FILENAME, .FNUM, FLAGS, SYNCDEPTH;

BEGIN
  INT ERROR := 1;

  WHILE ERROR DO
    BEGIN
      CALL OPEN ( FILENAME, FNUM, FLAGS, SYNCDEPTH );
      IF <> THEN
        BEGIN
          CALL FILEINFO ( FNUM, ERROR );
          RETURN ERROR;
        END;

         At this point, the file is open in the primary.

      IF ( STATUS := CHECKPOINT (STACKBASE, FNUM, 1) ) THEN
         CALL ANALYZE^CHECKPOINT^ERROR ( STATUS );

      CALL FILEINFO ( FNUM, ERROR );

      If this is executed because of a takeover from the
      primary, file-system error 16 (FILE NUMBER HAS NOT BEEN
      OPENED) is returned from the call to FILEINFO.  This
      will result in the "WHILE ERROR" loop being reexecuted.

    END;

  IF BACKUP^UP THEN
    BEGIN ! open the file in the backup.
      CALL CHECKOPEN(FILENAME,FNUM,FLAGS,SYNCDEPTH,,,ERROR);
      IF < THEN
        BEGIN ! backup exists, but could not open the file.
          CALL CLOSE ( FNUM );
          RETURN ERROR;
        END;
    END;

    RETURN 0; ! successful open by primary and backup if it
              ! exists.
 END; ! file open.
```

CREATING A DESCENDENT PROCESS OR PROCESS PAIR

As with opening files during processing, the possibility exists during creation of a descendant process or process pair that a failure can occur. This can result in the backup process creating a process already created by the primary. The following is a recommended method for descendant process creation:

```
CALL CREATEPROCESSNAME ( PNAME );

   The system generates a unique process name.

IF ( STATUS := CHECKPOINT(STACKBASE,PNAME,4) ) THEN
   CALL ANALYZE^CHECKPOINT^STATUS ( STATUS );
CALL NEWPROCESS ( PROGFILE,,, CPU, DESC^PID, ERROR, PNAME );
IF ERROR > 1 THEN
  IF ERROR.<0:7> <> 8        ! process name error.
     AND ERROR.<8:15> <> 10 ! can't communicate with sys mon !
     THEN BEGIN       ! unable to create the process due to
                      ! resource problem or coding error.
       .
       .
     END;
  ELSE
```

The following is necessary only if the backup needs the actual <cpu,pin> of the descendant process:

```
      BEGIN  ! duplicate name error, caused by takeover by
             ! backup.
        PPDENTRY ':=' PNAME FOR 3;
        CALL LOOKUPPROCESSNAME (PPDENTRY)
        IF < THEN ... ! process no longer exists.

        ! save descendant's process ID.
        DESC^PID ':=' PNAME FOR 4;

        ! determine actual <cpu,pin> of descendant.
        IF PPDENTRY[3].<0:7> <> CPU THEN
          IF PPDENTRY[4].<0:7> = CPU AND PPDENTRY[4] <> 0 THEN
             DESC^PID[3] := PPDENTRY[4]
          ELSE ... ! the process no longer exists in the CPU.
      END;
```

ADVANCED CHECKPOINTING

The following information is intended for application programmers who do not wish to use the checkpointing facility, but prefer instead to write their own checkpointing routines.

Backup Open

"Backup open" is a form of file open that permits a file to be open concurrently by both the primary and backup of a process pair regardless of the exclusion mode set by the primary process (except that access and exclusion modes must be the same for both the primary and the backup process, and file security is still enforced). This is accomplished by passing two parameters to OPEN: the process ID of the primary process which already has the file open, and the file number that was returned to the primary when it opened the file. After this form of OPEN, the primary and backup share access to the file such that in the case of disc files, when one process locks the file, the file becomes locked on behalf of both. (See Figure 12-4.)

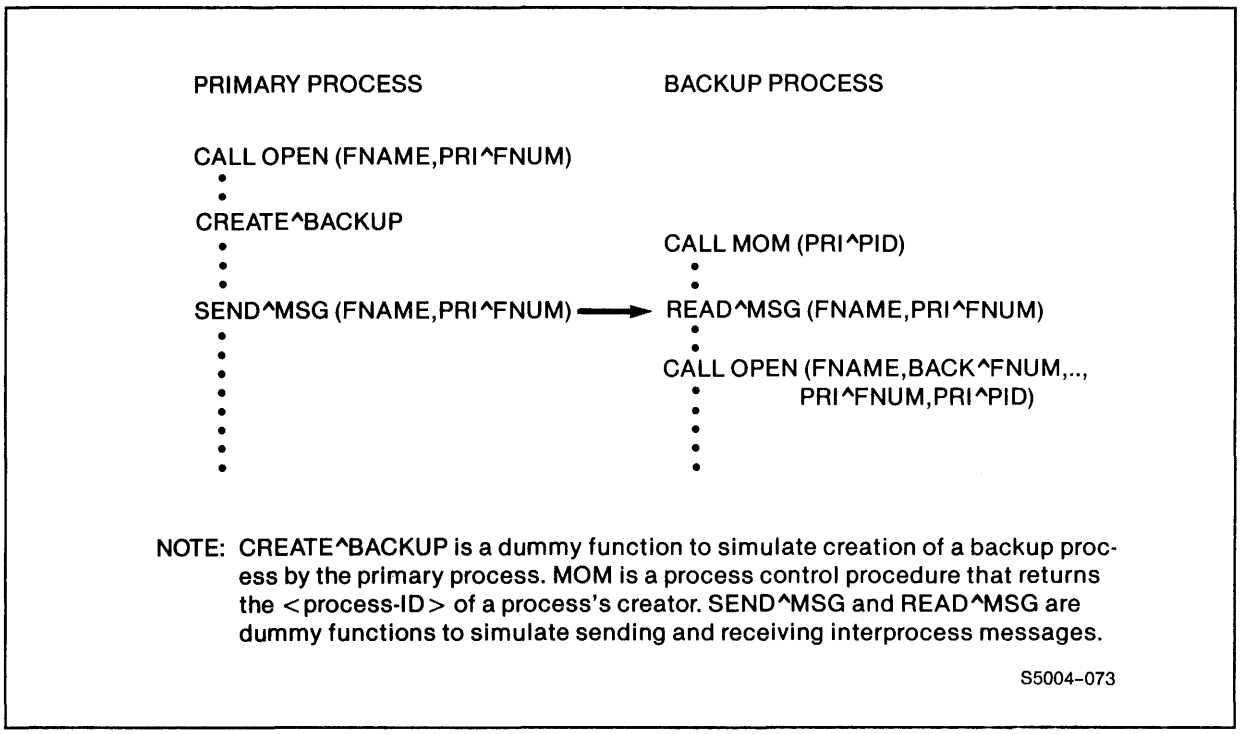


Figure 12-4. Backup Open by Backup Process

File Synchronization Information

File synchronization (sync) information is used by the system to determine if an operation by a backup process after a failure of its primary process is a new operation or a retry of an operation just performed by the primary.

The use of the sync information is accomplished in three parts:

1. Sync Depth

   The number of nonretryable operations that the file system is to "remember" is specified in the <sync-depth> parameter to the OPEN procedure. This normally is the number of write operations that a primary process performs to a file between checkpoint messages to its backup.

   The following is an example of opening a file and specifying a <sync-depth> of one:

   ```
   CALL OPEN (filename, filenum, ,1);
   ```

   If opened by the backup process of a process pair, the primary file number and process ID must also be specified.

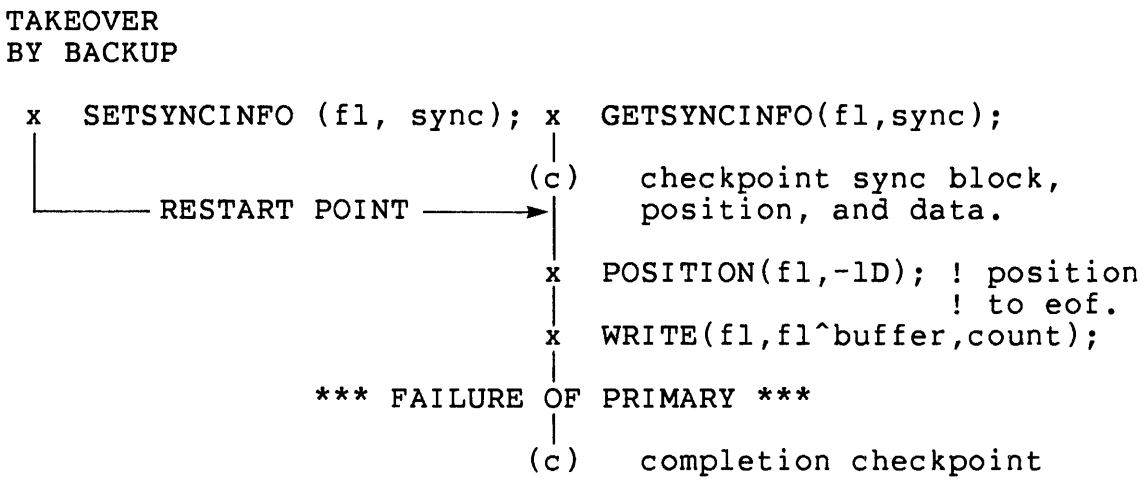
2. GETSYNCINFO Procedure

   When a primary application process is about to update a file by performing "sync depth" writes to the file, it first calls the GETSYNCINFO procedure, which returns "sync information" for the file. This information (which, incidentally, is never explicitly referenced by the application process) is then passed, along with the data to be written, in a checkpoint message to the backup application process. The primary process then performs the write operations, and upon completion informs its backup.

3. SETSYNCINFO Procedure

   If the primary application process fails, the backup process is notified by the operating system. Before attempting error recovery, the backup calls SETSYNCINFO with the sync information received in the latest checkpoint message. This synchronizes the retry operations that the backup is about to perform with any writes that the primary was able to complete before it failed. The backup then retries each write in the series (in the same order as the primary). If any operation is completed successfully by the primary, it is not performed by the file system; instead, just the completion status is returned to the backup process.

For example, in the following sequence of file-system operations, a call to GETSYNCINFO precedes the file operations, and a call to SETSYNCINFO precedes the restart point:



In this case, the write by the primary completed successfully, and the write by the backup when it takes over is ignored. The backup receives the completion status that the primary received prior to the primary's failure.

Another procedure, RESETSYNC, is provided for cases where, after a failure, the backup process wishes to execute its error recovery by performing different operations than those of the primary, or where the backup process does not have a current sync block and the operations performed by the primary are not known. In either case, it is undesirable to have the file system mistakenly relate an operation performed by the backup to a different operation which was performed by the primary. By calling RESETSYNC after taking over for the failed primary process, the backup process ensures that this does not occur.

A call to RESETSYNC causes a RESETSYNC system message (system message -34) to be sent to the paired-access process file referenced in the call, indicating that the sync ID for that file has been reset to 0. Upon receipt of this message (receipt of RESETSYNC messages must be enabled by setting OPEN <flags>.<1> = 1 when opening the file), a server process using the sync ID mechanism should clear its local copy of the sync ID value.

Advanced Usage of Checkpointing With TMF

Analysts and system programmers who wish to obtain further information on the use of TMF in conjunction with checkpointing, refer to the Tandem Journal, Volume 1, Number 1, Fall 1983.

SECTION 13

TRAPS AND TRAP HANDLING

During program execution, all error and exception conditions not related to input or output are handled by the trap mechanism. Conditions that are trapped typically are caused by coding errors in an application program or by a shortage of resources. For example, errors such as "arithmetic overflow" might be traceable back to user code, whereas an error such as "no memory available" might originate from the memory manager.

The default trap handler is DEBUG (or INSPECT, if specified for the process). In other words, if you do nothing, all traps result in control being passed to either DEBUG or INSPECT. (If INSPECT is specified for the process but is not available, DEBUG is used.)

If you choose to handle your own traps, the system procedure ARMTRAP is provided so you can specify the procedure that will handle the trap. When a trap occurs, control will be passed to your trap handler rather than to DEBUG or INSPECT. Your trap handler is notified of the particular trap condition. ARMTRAP is described in the System Procedure Calls Reference Manual.

The following pages describe the various trap conditions and include an example of how to write your own trap handler.

TRAP CONDITIONS

Certain critical error conditions occurring during process execution prevent the normal execution of a process. These errors, which are for the most part unrecoverable, cause traps to operating system trap handlers. The conditions are:

Trap no.	Description
0	Illegal address reference
1	Instruction failure
2	Arithmetic overflow
3	Stack overflow
4	Process loop-timer timeout
11 (%13)	Memory manager read error
12 (%14)	No memory available
13 (%15)	Uncorrectable memory error

- Illegal Address Reference--An address was specified that was not within either the virtual code area or the virtual data area allocated to the process. Virtual code area allocation is determined by the size of the program's code area. By default, virtual data area allocation is determined by the TAL compiler to be equal to the number of memory pages needed for the program's global storage plus one memory page for the program's data stack. The size of the virtual data area can be increased with the ?DATAPAGES command of the TAL compiler, the MEM parameter of the command interpreter RUN command, or the <memory pages> parameter of the NEWPROCESS procedure.

- Instruction Failure--An attempt was made to execute a code word that is not an instruction; an attempt was made by a nonprivileged process to execute a privileged instruction; or an illegal extended address reference was made.

- Arithmetic Overflow--The environment register overflow bit, ENV.<10>, is a 1 and the environment register traps enabled bit, ENV.<8>, is a 1. The overflow bit is set to 1 by the hardware if the result of a signed arithmetic operation could not be represented with the number of bits available for the particular data type. Arithmetic overflow also occurs if a divide with a divisor of zero is attempted. Note that the overflow bit in the ENV register is not automatically cleared. If the application process is to recover from the overflow condition, it must specifically clear the ENV register overflow bit (otherwise, another overflow trap will occur).

  The traps enabled bit of the ENV register is set to 1, by default, when a new process is created. A process can ignore a trap condition upon detection. See the example trap handler procedure.

- Stack Overflow--An attempt was made to execute a procedure or subprocedure whose (sub)local data area extends into the upper 32K of the data area. When calling an operating system procedure, stack overflow also occurs if there is not enough remaining virtual data space for the procedure to execute (the procedure does not execute). The amount of virtual data space available is the lesser of 'G'[32767] and the upper bound of

the process's virtual data area (the number of data pages specified when the process was created or run). Operating system procedures require approximately 350 words of user data stack space to execute.

- Process Loop-Timer Timeout--This occurs only if the process has enabled process loop timing by making a call to the SETLOOPTIMER process control procedure. This trap indicates that the new time limit specified in the latest call to SETLOOPTIMER has expired.

- Memory Manager Disc Read Error--This error indicates that a hard (unrecoverable) read error occurred while attempting to bring a page in from virtual memory.

- No Memory Available--This indicates that a page fault occurred, but no physical memory page is available for overlay.

- Uncorrectable Memory Error--This error indicates that an uncorrectable memory error was detected.

- If a trap has occurred and another trap occurs before the process can rearm its private trap handler with a call to ARMTRAP, the process is deleted and the creator of the process receives a process ABEND message.

- If a process uses the default trap handler DEBUG (or INSPECT), only the first trap will go to the trap handler. If the RESUME command is specified in DEBUG (or INSPECT) and a second trap occurs, the process is deleted and the creator of the process receives a process ABEND message.

Traps While Executing System Code

If a trap occurs while a process is executing in system code, the trap is deferred until the system procedure exits the system code and returns to the user environment. When control is passed to the trap handler, the location of the trap passed to the trap handler is the location of the call to the system procedure and the S register at the trap is -1 to signify that a deferred trap occurred while in system code. It is unlikely that a process can resume execution following a trap in system code because the correct S value is lost.

One exception is the process-loop-timer trap. If a process loop timer times out and causes a trap while in system code, the trap is deferred until the process returns to the user environment, but the value of S at this trap is not -1; in this case it is the

correct S value, which allows the process to easily resume execution following a loop-timer timeout.

DEFAULT TRAP HANDLER

If no trap handler is specified (ARMTRAP is not called), then DEBUG or INSPECT will be the default trap handler.

USER-DEFINED TRAP HANDLER

The ARMTRAP procedure is used to specify an entry point into the application program where execution is to begin if a trap occurs. The syntax for the ARMTRAP procedure and considerations for its use are presented in the System Procedure Calls Reference Manual.

EXAMPLE

The following is an example of an application procedure that displays the current value of the P register when an arithmetic overflow trap occurs. Following an arithmetic overflow trap, the trap mechanism is rearmed, and the application process continues processing. If any other trap occurs, the procedure calls the DEBUG procedure.

The example trap handler procedure is:

```
PROC  OVERFLOWTRAP;
  BEGIN
    INT  REGS = 'L'+1,          ! R0-R7 saved here.
         WBUF = 'L'+9,          ! buffer for terminal I/O.
         PREG = 'L'-2,          ! P register at time of trap.
         EREG = 'L'-1           ! ENV register at time of trap.
         TRAPNUM = 'L'-4;       ! trap number.
         SPACEID = 'L'-5;       ! spaceid of trap location.
    DEFINE  OVERFLOW = <10>#;   ! overflow bit in ENV register.
    STRING  SBUF = WBUF;        ! string overlay for I/O buffer
    LITERAL  LOCALS = 15;       ! # of words of local storage.

    ! arm the trap.
    CALL  ARMTRAP( @TRAP, $LMIN ( LASTADDR, %77777 ) - 500 );
    RETURN;

    ! enter here on a trap,
    ! save R0-R7 and allocate local storage.

TRAP:
    CODE (  PUSH %777;  ADDS LOCALS  );

    ! call DEBUG if the trap is not an overflow condition.

    IF  TRAPNUM <> 2  THEN CALL DEBUG;

    ! format and print the message on the home terminal.

    SBUF ':=' "ARITHMETIC OVERFLOW AT %";
    CALL  NUMOUT( SBUF[24], PREG, 8, 6 );
    CALL  WRITE( HOME^TERM, WBUF, 30 );
    IF  <>  THEN ...

    ! the overflow bit must be cleared before the old values
    ! of the registers are restored.

    EREG.OVERFLOW := 0; ! clear overflow.
    CALL  ARMTRAP ( 0, $LMIN ( LASTADDR, %77777 ) -500 );

  END;
```

At the beginning of the program, the procedure is called:

```
CALL OVERFLOWTRAP;
```

From this point on, any arithmetic overflows are logged on the home terminal. For example, the following statement would cause the trap handler to be entered:

```
I := I/J;
```

if the current value of J were zero.

SECTION 14

USING EXTENDED MEMORY SEGMENTS

The extended addressing and memory management capabilities of the Tandem system are accessible through the GUARDIAN operating system procedures discussed in this section. These procedures allow a process to allocate, use, and deallocate extended data segments and to define and use memory pools in either the data stack or in extended data segments.

The procedures for using extended data segments are as follows:

ALLOCATESEGMENT	allocates an extended data segment for use by the calling process.
DEALLOCATESEGMENT	deallocates an extended data segment.
USESEGMENT	selects a particular extended data segment for use.
DEFINEPOOL	designates a portion of a user's stack or an extended data segment for use as a pool.
GETPOOL	obtains a block of memory from a pool.
PUTPOOL	returns a block of memory to a pool.

These procedures cannot be used on the Tandem NonStop 1+ system.

These procedures are described in detail in the System Procedure Calls Reference Manual.

EXTENDED MEMORY

The space normally available to a process for data is the process's user data, or stack, segment. The GUARDIAN operating system automatically allocates this segment at the time the process is created. It consists of up to 64K (65,536) words, or 128K bytes, of memory and can be accessed using either 16-bit addresses or 32-bit extended addresses.

To obtain larger areas of memory for data, a process can use a set of GUARDIAN procedures to explicitly allocate, use, and deallocate memory in the form of extended data segments. These segments can be as large as 128 megabytes (134,217,728 bytes). Extended data segments are discussed more fully in the System Description Manual.

To request allocation of an extended data segment, a process calls the ALLOCATESEGMENT procedure. Once the segment has been allocated successfully, it must be put in use by a call to the USESEGMENT procedure before it can be accessed. When the segment is no longer needed, it can be freed by calling the DEALLOCATESEGMENT procedure.

Extended data segments are accessed by means of 32-bit extended addresses (extended pointers), as described in the Transaction Application Language (TAL) Reference Manual. Extended data segments may also be used in COBOL or BASIC, which invoke the GUARDIAN procedures and access the extended memory automatically; refer to the COBOL Programming Manual or the Tandem EXTENDED BASIC Reference Manual when using extended memory in these languages.

A user process can have several extended data segments, each referred to by a different segment ID number supplied by the process when it requests allocation of the segment. However, a process can have only one extended data segment in use at a time.

Processes in the same CPU may share extended data segments. An extended data segment is first allocated by one process by calling ALLOCATESEGMENT without the <pin> parameter. Subsequent calls to ALLOCATESEGMENT by other processes should specify both the <pin> of the process that owns the segment and the same <segment-id> for the segment. The sharing process must have access rights as described with the ALLOCATESEGMENT procedure syntax in the System Procedure Calls Reference Manual. Once a segment is shared by two or more processes, the segment will not be deallocated until all owning processes have deallocated the segment or stopped.

DYNAMIC MEMORY ALLOCATION

Three GUARDIAN procedures, DEFINEPOOL, GETPOOL, and PUTPOOL, support the creation of memory pools and dynamic allocation of variable-sized blocks from the pool. The calling program provides the memory area to be used as the pool and then calls the DEFINEPOOL procedure to initialize a 19-word array, called the pool header, which is used to manage the pool. The pool and the pool header can reside in the user data stack or in extended memory. The pool routines accept and return extended addresses that apply to both the stack and extended memory.

Once the pool is defined, the process can reserve blocks of various sizes from the pool by calling the GETPOOL procedure and release blocks by calling the PUTPOOL procedure. The program must release one entire block using PUTPOOL; it may not return part of a block or multiple blocks in one PUTPOOL call.

If the process uses anything other than the currently reserved blocks of the pool, the pool structure will be corrupted and unpredictable results will occur. If multiple pools are defined, reserved blocks must be returned to the right pool. For debugging purposes, a special call to GETPOOL is provided that checks for pool consistency.

Pool Management Methods

The following information is supplied for use in evaluating the appropriateness of using GUARDIAN's pool routines in user application programs and to determine the proper size of a pool. Application programs should not depend on the pool data structures since they are subject to change, but should use only the three pool procedural interfaces described above.

These procedures are described in detail in the System Procedure Calls Reference Manual.

The requested block size is rounded up to a multiple of 4 bytes and to a minimum of 28 bytes. This reduces pool fragmentation, but wastes memory space when the program is allocating small blocks.

Two extra words are allocated for tags at the beginning and end of each block; thus, the minimum pool block size is 32 bytes. These tags serve three purposes: (1) they contain the size of each block so that the program need not specify the length of the block when releasing it, (2) they serve as a check that the program has not erroneously used more memory than the block

contains, and (3) they provide for very efficient coalescing of adjacent free blocks.

In GETPOOL, the free block list is searched for the first block sufficiently large to satisfy the request. If the free block is at least 32 bytes longer than the required size, it is split into a reserved block and a new free block. Otherwise, the entire free block is reserved for the request.

In summary, the pool space overhead on each block can be substantial if very small blocks are being allocated. An exact formula is:

```
BYTES^ALLOCATED := ( $MAX ( REQUEST + 7, 32 ) / 4) * 4
```

where REQUEST is the original request size in bytes.

Although they can also be used to manage the allocation of a collection of equal-size blocks, these procedures are not recommended for that purpose because they will consume more CPU time and pool memory than routines designed for that specific task.

SECTION 15

ADVANCED USES OF MEMORY

This section describes memory management techniques that can affect the efficiency of a process. Incorrect use of these techniques could result in degradation of system performance.

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 15-1). 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 LCBs is required for each message transfer that is in progress at any given moment. Additionally, 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.

The reserved LCBs feature is used so that an application process will not be suspended while waiting for an LCB to be allocated.

The application process can reserve LCBs by calling the RESERVELCBS procedure. This requires specifying the number of LCBs to be reserved for receiving messages (receive LCBs) and the number to be reserved for sending messages (send LCBs):

- Receive LCBs

  On the receiving side of a message transfer, one receive LCB is used for each incoming message that is queued on an application process's $RECEIVE file. When the $RECEIVE file is opened with a nonzero receive depth, an LCB is also required for each message that the application process has read using READUPDATE but has not replied to.

  The receiving application process can guarantee that it will never miss any messages by reserving one LCB for each possible incoming message that can be queued.

- Send LCBs

  On the sending side of a message transfer, one send LCB is used for each outgoing message. If no send LCB is available or no receive LCB is available when a process wishes to send a message, the process will be suspended until LCBs become available or a timeout occurs. If a timeout occurs, then file-system error 30 is returned, and an operator message is logged on the console.

  The sending application process can guarantee that it will never be suspended for a send LCB allocation by reserving one send LCB for each outgoing message that can be outstanding at any given moment. One message is outstanding for each outstanding nowait I/O operation, plus one message can be outstanding for a wait I/O operation or call to a process control procedure.

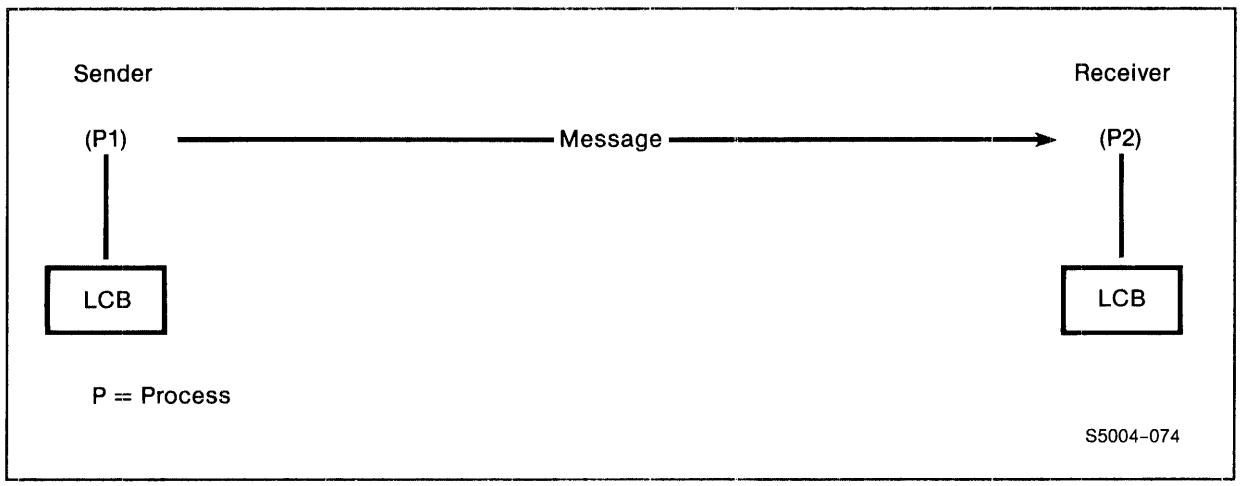


Figure 15-1. Link Control Blocks

SECTION 16

MISCELLANEOUS UTILITY PROCEDURES

The procedures included in this section fall into categories not clearly defined in other sections of this manual. Procedures are grouped together here in terms of time, string and number manipulation, and other utility procedures.

PROCEDURES OVERVIEW

The following utility procedures relate to time and the system clock:

JULIANTIMESTAMP returns a four-word, GMT, Julian-date based timestamp.

COMPUTETIMESTAMP converts a Gregorian date and time of day into a 64-bit timestamp.

INTERPRETTIMESTAMP converts a 64-bit Julian timestamp into a Gregorian date and time of day, and into a Julian day number.

CONVERTTIMESTAMP converts a Greenwich mean time (GMT) to or from a local timestamp within any accessible node (at GUARDIAN level B00 or greater) in the network.

COMPUTEJULIANDAYNO converts a Gregorian calendar date to a Julian day number.

INTERPRETJULIANDAYNO converts a Julian day number to the Gregorian calendar year, month, and day.

SETSYSTEMCLOCK allows you to change the system clock time if you are the super ID (255.*).

TIME provides the current date and time in integer form.

CONTIME takes 48 bits of a timestamp and provides a date and time in integer form.

TIMESTAMP provides the current value of the processor clock where this application is running.

In addition, procedures are provided for the timing of processes. They are described under "Process Timing" in Section 3.

The following utility procedures pertain to string and number manipulation:

SHIFTSTRING upshifts or downshifts alphabetic characters in a string.

FIXSTRING edits a string of characters based on information supplied in an editing template.

NUMIN converts the ASCII representation of a number into its internal machine (binary) equivalent.

NUMOUT converts the internal machine representation of a number to its ASCII equivalent.

HEAPSORT sorts an array of equal-sized elements in place.

The following utility procedures are also available:

INITIALIZER reads the startup message and, optionally, the assign and param messages to prepare global tables and initialize file control blocks (FCBs).

LASTADDR provides the global ('G'[0] relative) address of the last word in the caller's data area.

TOSVERSION provides the identifying letter and number indicating which GUARDIAN operating system version is running in this system.

REMOTETOSVERSION supplies the identifying letter and number indicating which GUARDIAN operating system version is running in a particular system of the network.

SYSTEMENTRYPOINTLABEL returns either the procedure label of the named entry point, or zero if the entry point is not found.

The syntax and considerations for using these procedures are presented in the System Procedure Calls Reference Manual.

PROCEDURES RELATED TO TIME

It is extremely important to coordinate the activities of the various processors in a system based on time. Originally, this coordination was achieved by advancing the time of other processor clocks to match that of the fastest clock in the system. As a result, system time typically ran faster than wall-clock time by a few seconds each day. Accumulated gains in time accrued under this method were then discarded by the operator (using the SETTIME procedure) at convenient junctures such as cold loads, weekend time resets, and so on.

The need for exact system time clocks increased with network usage and the opportunities to reset all system clocks diminished. It was not always acceptable to run all clocks at the speed of the fastest processor clock in the system, nor was it acceptable to have the system clock move backwards.

The time procedures provide four-word, microsecond-resolution timestamps, Julian date based timestamps; CPU clock-rate averaging; clock-rate adjustment; automatic daylight savings time adjustments; Julian date conversion routines; and a callable procedure to set the system clock programmatically. These procedures affect:

- how clocks are set
- how clocks are synchronized
- how the user obtains and interprets timestamps
- how processes can obtain clock adjustment information.

Clock Setting

Originally, the GUARDIAN command interpreter command SETTIME was the only tool available to the system operator (group ID = 255) to correct system clocks. The SETTIME command still exists.

A callable procedure, SETSYSTEMCLOCK, now allows any system operator or super ID to set the system clock.

Clock Averaging

Clocks in a given system are now averaged instead of synchronized with the fastest clock in the system. This scheme greatly diminishes the cumulative effects of keeping up with the fastest clock. Small adjustments to system clock time are performed by software clock rate adjustment.

Terms

Greenwich mean time (GMT)	Popular basis for calculating time throughout the world. Based on mean solar time for the meridian at Greenwich, England. Also known as Coordinated Universal Time.
Local standard time (LST)	Does not include daylight savings time.
Local civil time (LCT)	Includes daylight savings time.
Daylight savings time (DST)	Extends the amount of daylight by changing local civil time. In the United States, DST advances all clocks one hour at 02:00 hours LST on the last Sunday in April and reverts to standard time at 02:00 hours LST on the last Sunday in October.
Julian day number (JDN)	The integral number of days since January 1, 4713 B.C.
Gregorian calendar	The common civil calendar. The Gregorian calendar was sponsored by Pope Gregory XIII. It has been adopted by most countries of the world.

JULIANTIMESTAMP Procedure

The JULIANTIMESTAMP procedure returns a four-word, microsecond resolution, GMT, Julian date based timestamp. The returned value represents the number of microseconds since noon of January 1, 4713 B.C.

NOTE

The RCLK instruction should not be used in applications requiring time of day. All software that performs RCLK instructions for the purpose of obtaining a timestamp should be changed to use the JULIANTIMESTAMP procedure. Elapsed-time measurements can still safely use the RCLK instruction.

COMPUTETIMESTAMP, CONVERTTIMESTAMP, and INTERPRETTIMESTAMP

The COMPUTETIMESTAMP procedure converts a Gregorian date and time of day into a 64-bit timestamp. The COMPUTETIMESTAMP array is an array containing a date and the time of day. This 8-word array has the following form and bounds:

<date-n-time>	[0] =	the Gregorian year, such as 1985	(1-4000)
	[1] =	the Gregorian month	(1-12)
	[2] =	the Gregorian day of month	(1-31)
	[3] =	the hour of the day	(1-23)
	[4] =	the minute of the hour	(0-59)
	[5] =	the second of the minute	(0-59)
	[6] =	the millisecond of the second	(0-999)
	[7] =	the microsecond of the millisecond	(0-999)

To obtain a timestamp, you might use:

```
TS := COMPUTETIMESTAMP (DATE^N^TIME, ERRORMASK);
```

If the ERRORMASK parameter is supplied, each element of DATE^N^TIME is checked for validity. A bit value of 1 indicates that the element is valid.

The INTERPRETTIMESTAMP procedure converts a 64-bit Julian timestamp into a Gregorian (common civil calendar) date and time of day. It also returns (as its value) the 32-bit Julian day number. No checking for the range of the Julian timestamp is

performed. The caller must ensure that the Julian timestamp corresponds to time in the range of 01 January 0001 00:00 to 31 December 4000 23:59:59.999999

The CONVERTTIMESTAMP procedure converts a GMT timestamp to or from a local timestamp within any accessible node in the network.

COMPUTEJULIANDAYNO and INTERPRETJULIANDAYNO Procedures

The COMPUTEJULIANDAYNO procedure computes the Julian day number from a Gregorian calendar date on or after January 1, 0001. The Gregorian calendar date must be valid.

The Gregorian calendar date is written as year, month, and day. For example, to find the Julian day number for the Gregorian calendar date January 9, 1985, you might use the following:

```
JDN := COMPUTEJULIANDAYNO (1985, 01, 09, ERROR);
```

The INTERPRETJULIANDAYNO procedure converts a Julian day number to the Gregorian calendar year, month, and day. The Julian day number to be converted must be equal to or greater than 1,721,426 (which is the same as 0001 January 01, Gregorian).

SETSYSTEMCLOCK Procedure

The SETSYSTEMCLOCK procedure allows the super ID (privileged) caller to change the system clock time by means of a callable procedure.

NOTE

The following three procedures: TIME, CONTIME, and TIMESTAMP, are retained for compatibility with previous systems.

TIME Procedure

The TIME procedure provides the current date and time in integer form. Use the TIME procedure to determine the effects of the SETTIME command or the SETSYSTEMCLOCK procedure.

CONTIME Procedure

The CONTIME procedure converts a 48-bit timestamp to a date and time in integer form.

Use CONTIME to return an array of the date and time in seven separate segments, from calendar year to .01 seconds.

To use CONTIME to convert the <last-mod-time> timestamp into a readable form, you can use:

```
INT LAST^T[0:2], DATE^TIME[0:6];
```

Then the last modification time is obtained through a call to the FILEINFO procedure:

```
CALL FILEINFO( FNUM,,,,,,,,LAST^T);
```

Then CONTIME is used to convert the three words in <LAST^T> to a date and time:

```
CALL CONTIME(DATE^TIME,LAST^T,LAST^T[1],LAST^T[2]);
```

Seven words of date and time are returned in <DATE^TIME>.

TIMESTAMP Procedure

The TIMESTAMP procedure provides the current value of the processor clock where this application is running.

PROCEDURES FOR STRING AND NUMBER MANIPULATION

A number of procedures can be considered primarily of use in manipulating strings, numbers, and arrays.

SHIFTSTRING Procedure

The SHIFTSTRING procedure upshifts or downshifts all alphabetic characters in a string. Non-alphabetic characters remain unchanged. See the SHIFTSTRING procedure in the System Procedure Calls Reference Manual.

Typically, commands entered interactively from a terminal are upshifted to ensure that they parse properly and compare easily. The SHIFTSTRING procedure is used to upshift all lowercase alphabetic characters to uppercase. The <casebit> parameter of the SHIFTSTRING procedure indicates whether to upshift or downshift the string. If bit 15 is 0, all alphabetic characters are upshifted; if bit 15 is 1, they are downshifted. For example, the following line upshifts all alphabetic characters in a 32-byte string whether they were originally in uppercase, lowercase, or any combination thereof:

```
CALL SHIFTSTRING ( STRINGB, 32, 0 )
```

FIXSTRING Procedure

The FIXSTRING procedure is used to edit a string based on subcommands provided in a template. The FIXSTRING procedure is commonly used to implement an FC command in an interactive process. Use of the FC or "Fix" command (to correct typing errors in interactive commands) is fully explained in the GUARDIAN User's Guide.

See the FIXSTRING procedure in the System Procedure Calls Reference Manual for an explanation of its syntax.

FIXSTRING was used to implement the FC command in the DEBUG facility and in the GUARDIAN command interpreter.

If you wish to implement your own FC command, the following example illustrates how it can be implemented in an interactive command interpreter:

```
.
INT .command[-1:3] := "< ",  ! command length <= 8 characters
    .last^command[0:3],      ! save previous command
     num,                    ! length of current command
                             ! string
     save^num;               ! length of last command string

STRING .scommand := @command '<<' 1;  ! command addressed as
                                       ! string
```

```
  .
  .
INT PROC fc; FORWARD;

  .
  .
PROC command^interpreter;
  BEGIN
      .
      .
    INT repeat := 0;     ! a flag used to determine whether
                         ! command^interpreter should attempt
                         ! to execute a command upon return
                         ! from "fc"
      .
      .
    WHILE 1 DO    ! the main loop of command^interpreter
                  ! executes until an "exit" command is
                  ! encountered.
      BEGIN

        IF NOT repeat THEN
          BEGIN
            command ':=' "< ";  ! assume "<" is the prompt
                                ! character
            CALL WRITEREAD(term, command, 1, 8, num);

               Displays the prompt character and reads a
               command, assuming "term" is the device number
               of the terminal.

          END;

        IF command = "FC" THEN repeat := fc
        ELSE
          BEGIN      ! identify and execute command,
                     ! or print an "illegal command" message.
            .
            repeat := 0;
          END;

        IF num THEN
          BEGIN
            save^num := num;
            last^command ':=' command FOR (save^num+1)/2;

               Saves last command and its length in case next
               command is "FC".

           END;
      END;         ! main loop
  END;             ! command^interpreter
```

```
.
.
INT PROC fc;
  BEGIN
    INT .temp^array[0:35],    ! array to hold modification
                              ! template
        temp^len;             ! length of template
    STRING .s^temp^array := @temp^array '<<' 1;
                              ! temp^array addressed as string

    command[-1] := "< ";
    num := save^num;
    command ':=' last^command FOR (num+1)/2;

    DO
      BEGIN

        CALL WRITE(term, command[-1], num + 2 );
                ! display "<" followed by the last command.
        temp^array := " .";  ! template prompt
        CALL WRITEREAD ( term, temp^array, 2, 72, temp^len );
                         ! display prompt and read template.
        IF > OR temp^len = 2 AND temp^array = "//" THEN
          BEGIN      ! restore command
            num := save^num;
            command ':=' last^command FOR (num+1)/2;
            RETURN 0;
          END;

           An EOF or a template consisting of "//" causes "fc"
           to return 0, indicating that "command^interpreter"
           should not execute the command, but should prompt
           for a new command instead.  If the new command is
           "FC", then the string to be fixed is the command
           that was originally being modified on the previous
           call to "fc".

        CALL FIXSTRING (s^temp^array,temp^len,scommand, num);

           "scommand" now contains the modified command, and
           "num" is its length.  If "temp^len" > 0, the loop
           executes again, displaying the modified command and
           expecting a new template.  If "temp^len" = 0, then
           a <cr> was input instead of a template.  In this
           case, FIXSTRING leaves the command unchanged and
           returns a value of 1, indicating that the command
           interpreter should attempt to identify and execute
           the command.

      END
    UNTIL NOT temp^len;  ! loop executes until "temp^len" = 0,
                         ! indicating a <cr>
```

```
    RETURN 1;    ! indicates to the calling procedure that
                 ! the command came from "fc" and should be
                 ! identified and executed.
END;  !fc
```

NUMIN and NUMOUT Procedures

The NUMIN function procedure converts ASCII representations of numbers, from base 2 through base 10, to signed integer values. The NUMOUT procedure converts unsigned integer values to their ASCII equivalents using any number base from 2 through 10. The result is returned right-justified in an array, filled with leading zeros. Refer to the NUMIN and NUMOUT procedures in the System Procedure Calls Reference Manual.

If you want to convert a binary, octal, or decimal number to an INT value, consider using the NUMIN procedure.

Examples:

The value of NUMIN can be used to determine the number of characters converted:

```
STRING number [0:9] := "12345     ";
INT result, status, .next^char
```

Then NUMIN is invoked:

```
@next^char := NUMIN( number , result , 10 , status );
```

After NUMIN executes, the pointer variable "next^char" contains the address of "number[5]" (the sixth element).

Then subtracting

```
num^converted := @next^char '-' @number;
```

provides the number of characters used in the conversion (in this example, five).

An alternate way of doing the same:

```
num^converted := NUMIN(number,result,10,status) '-' @number;
```

Another example, this time showing a string containing an ASCII number greater than the base being converted:

```
STRING number[0:5] := "%19234";
```

Then NUMIN is invoked:

```
@next^char := NUMIN(number, result, 8, status);
```

The only character converted to its octal representation is "1". At completion, the pointer variable "next^char" points to the character "9".

The NUMOUT procedure can be used to reverse the process. For example, if you want to convert an INT value to its base-10 ASCII equivalent, use the following:

```
STRING array[0:5];
INT variable := 2768;
LITERAL base = 10, width = 6;
   .
   .
CALL NUMOUT(array, variable, base, width);
```

After NUMOUT executes, ARRAY contains:

```
"002768"
```

Another example, using the same number but converting to base 8:

```
   .
   .
CALL NUMOUT(array, variable, 8, width);
   .
```

After NUMOUT executes, ARRAY contains:

```
"005320"
```

A final example, using the same number and converting to base 10 but with a "width" of 3:

```
   .
CALL NUMOUT( array, variable, 10, 3);
   .
```

After NUMOUT executes, "array" contains:

```
"768"
```

The result is truncated to three characters; the three leftmost characters are lost.

HEAPSORT Procedure

The HEAPSORT procedure is used to sort an array of equal-sized elements in memory. It is only used to sort memory arrays. It cannot be used with extended addresses.

You can use HEAPSORT to sort variable-size strings by sorting an array of pointers to the variable-size strings and using a suitably clever comparison procedure. If the strings won't all fit in memory, you must put them in a file and use the SORT program.

See the SORT/MERGE User's Guide for the more common methods of sorting, such as sorting files.

Refer to the HEAPSORT procedure in the System Procedure Calls Reference Manual for an explanation of its syntax.

The following example illustrates the use of HEAPSORT.

```
LITERAL element^size = 12;

INT .array[0:119], ! array to be sorted.
    num^elements;
```

Elements of twelve words each are to be sorted in ascending order. Therefore, the compare procedure can be written:

```
INT PROC ascending (a,b);
   INT .a, .b;
  BEGIN
    RETURN IF a < b FOR 12 THEN 1 ELSE 0;
  END;
```

Then HEAPSORT is called to sort ARRAY:

```
    .
num^elements := 10;
CALL HEAPSORT(array,num^elements,element^size,ascending);
    .
```

sorts the ten elements in ARRAY in ascending order.

OTHER PROCEDURES

The following procedures, especially the INITIALIZER procedure, are quite useful, but are not related to other procedures.

INITIALIZER Procedure

The INITIALIZER procedure provides a way to receive startup, assign, and param messages and yet allows you to ignore the details of $RECEIVE protocol. (The INITIALIZER obtains messages from $RECEIVE and calls the user-supplied procedure, passing the messages as a parameter to the procedure.) See the description of the INITIALIZER procedure in the System Procedure Calls Reference Manual.

Also, if you supply the <RUCB> parameter, the INITIALIZER will store data into FCBs based on the information supplied by the startup and assign messages. These FCBs are in the form expected by the sequential I/O procedures, and may be used with the sequential I/O procedures without change. If your application does not use the sequential I/O procedures to access the files, the information recovered from the assign messages can be obtained from the FCBs by using the SET^FILE procedure. See Section 17, "Using the Sequential I/O Procedures".

The INITIALIZER reads the startup message, then optionally requests assign and param messages. For each assign message the FCBs (if <RUCB> is passed) are searched for a logical file name matching the logical file name contained in the assign message. If a match is found, the information from the assign message is put into the file's FCBs, and the match count is incremented. For proper matching of names, the "progname" and "filename" fields of the assign message must be blank filled.

The INITIALIZER is useful in program startup. It does the following:

In the primary process (<status> = 0):

1. Inspects <flags>.<13:15>, and calls the appropriate procedures, if any.

2. Opens $RECEIVE.

3. Reads the startup sequence from MOM:

    a. Stores startup and assign information in <RUCB> if an array was passed.
    b. Calls procedures if any are passed (optionally, assign and params messages).
    c. Calls ABEND if the messages read from $RECEIVE are not in the correct order. The correct order is open, startup message, param and/or assign messages (in any order), and close. If the creator does not send a CLOSE, INITIALIZER waits 60 seconds and calls ABEND.

d. Rejects messages from anyone other than MOM with reply code 100 (OPEN messages) or 60 (all others).

NOTE

If bit <11> of the flags word = 1, INITIALIZER is not to request assign and param messages from the parent process. In this case, it replies to the startup message with error return of zero and no data. If bit <11> is 0, INITIALIZER requests assign and param messages, depending on how the startup message was sent. If the parent sent the startup message with a WRITE, INITIALIZER replies with error 70 and no data. If the parent sent a startup message with WRITEREAD and nonzero <read-count>, INITIALIZER replies with error return zero and four bytes (32 bits) of data in which only the first two bits can be set. Bit 0 indicates assign messages; bit 1 indicates param messages. See Section 5, "Interfacing to the GUARDIAN command interpreter", for more information on the assign and param messages.

4. Closes $RECEIVE.

5. Performs the following:

   a. Substitutes the FCBs actual file names for default physical file names.
   b. Expands partial file names in the FCBs.
   c. Places information into the <RUCB> if used.
   d. Replaces system names with system numbers.

6. Returns 0, indicating primary process.

In the backup process (<status> = -1):

1. Inspects <flags>.<13:15> and calls the appropriate procedures, if any.

2. Calls CHECKMONITOR. If CHECKMONITOR returns, this indicates that the primary process failed before it made a stack checkpoint.

   In this case, if <flags>.<12> = 0, the INITIALIZER calls ABEND; if <flags>.<12> = 1, the INITIALIZER returns -1, indicating the CHECKMONITOR failed.

CHECKMONITOR does not normally return. For an overview of the checkpointing process, refer to Section 12.

LASTADDR Procedure

The LASTADDR (last address) function procedure returns the 'G'[0] relative address of the last word in the application process's data area. See the LASTADDR procedure in the System Procedure Calls Reference Manual.

The LASTADDR function can be used to determine the number of memory pages allocated to a running application program:

```
num^pages := LASTADDR.<0:5> + 1;
```

A bit extraction is performed on the six high-order address bits returned from LASTADDR; one is added to that value (figure 16-1).

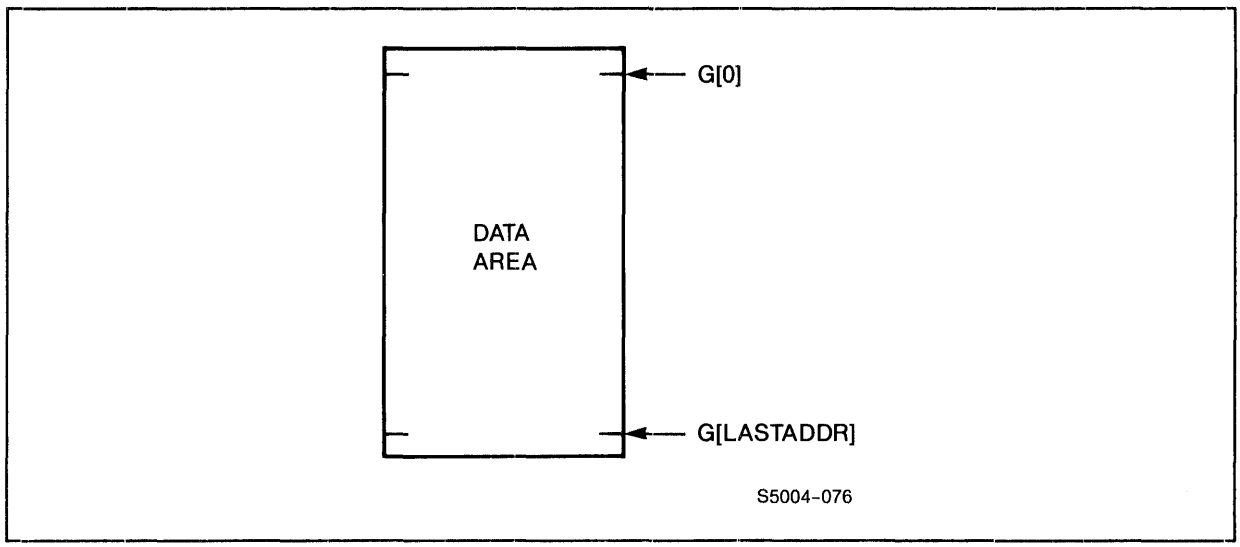


Figure 16-1. Last Address

SYSTEMENTRYPOINTLABEL Procedure

The SYSTEMENTRYPOINTLABEL procedure returns the procedure label of the named entry point. If the entry point is not found, a zero is returned.

TOSVERSION and REMOTETOSVERSION Procedure

The TOSVERSION procedure provides an identifying letter and number indicating which version of the GUARDIAN operating system is running.

If you want your application program to check the version of the GUARDIAN operating system currently running, use the TOSVERSION procedure call. The system level (the ASCII letter) and revision level (in binary) are returned.

The REMOTETOSVERSION procedure obtains the identifying letter and number indicating which version of the GUARDIAN operating system is running in a particular system of the network. Details of its use appear in the System Procedure Calls Reference Manual.

SECTION 17

SEQUENTIAL INPUT/OUTPUT PROCEDURES

The sequential I/O procedures were created to provide TAL programmers with a small, standardized set of procedures for reading and writing sequential files. These procedures were created as an addition to the GUARDIAN operating system procedures. They are described separately in the System Procedure Calls Reference Manual.

The primary benefit of these procedures is that programs using them can treat different file types in a consistent and predictable manner. The sequential I/O procedures (or SIO procedures, as they are commonly called) are recommended for most programs requiring sequential access to files. The SIO procedures are not difficult to learn to use, especially if you follow the examples provided later in this section.

If you need to write to files in EDIT format, the SIO procedures provide the only programmatic method to do so.

NOTE

> If you have not used the SIO procedures before, it is recommended that you follow the examples given in this section before studying the SIO procedure syntax. If you are already familiar with the use of the SIO procedures, you can work directly from the System Procedure Calls Reference Manual for your system.

Do not intermix GUARDIAN procedures with the SIO procedures on the same file. Though the procedure names appear compatible, in most cases they are not, and cause processes to ABEND.

The SIO procedures are especially useful because:

- They are essentially device-independent.
- They use accepted defaults for all operations.
- They can perform all common input and output operations.
- They can handle BREAK from a terminal.
- They can provide blocked I/O.
- They provide simplified error handling.
- They provide uniform sequential access to all types of files, including files in EDIT format.

If you want to write programs with the greatest ease and fewest problems, consider using the SIO procedures in combination with the INITIALIZER. INITIALIZER reads startup messages, assign messages, and param messages, and initializes in and out files in a Tandem standard manner.

Characteristics of the SIO procedures include:

- All file types are accessed in a uniform manner. File access characteristics, such as access mode, exclusion modes, and record size, are selected according to device type and intended access. The SIO procedures' default characteristics are set to facilitate their most common uses.
- Error recovery is automatic. Each fatal error causes a comprehensive error message to be displayed, all files to be closed, and the process to abort. Both the automatic error handling and the display of error messages can be turned off so your program can do the error handling.
- The SIO procedures can be used with the INITIALIZER procedure to make run-time changes by using the ASSIGN command. File transfer characteristics, such as record length, can be changed using the ASSIGN command. You can also assign a physical file name to a logical file. (See "Interface with INITIALIZER and ASSIGN Messages" later in this section.)
- The SIO procedures retain information about the files in file control blocks (FCBs). There is one FCB for each open file, plus one common FCB that is linked to the other FCBs.
- The SIO OPEN^FILE procedure lets an application alter certain access characteristics when a file is opened. These characteristics determine the nature of subsequent SIO

operations on the opened file. The SET^FILE procedure allows similar alterations either before or after the file is opened.

- SIO opens all files with a sync depth of 1.

Some features of the SIO procedures include:

- Individualized BREAK handling routines
- Error handling routines that allow retryable errors
- Interprocess I/O
- Automatic file creation
- Trimming trailing blanks
- Truncating lines
- Automatic top-of-form control
- Carriage return (CR) or line feed (LF) on BREAK

The SIO I/O procedures and their functions are:

CHECK^BREAK	checks whether the BREAK key was pressed.
CHECK^FILE	retrieves file characteristics.
CLOSE^FILE	closes a file.
GIVE^BREAK	disables BREAK processing by the process.
NO^ERROR	allows manual handling of error messages.
OPEN^FILE	opens a file for access by the SIO procedures.
READ^FILE	reads from a file.
SET^FILE	sets or alters file characteristics.
TAKE^BREAK	enables BREAK processing by the process.
WAIT^FILE	waits for the completion of an outstanding I/O operation.

All of the SIO procedures and their parameters, default values, and flag settings, are described in the System Procedure Calls Reference Manual. Examples using these SIO procedures appear later in this section.

FCB STRUCTURE

File characteristics and procedure call information are kept in file control blocks (FCBs) within the user data space. The SIO FCBs must be created by you--they are maintained in addition to the system FCBs created and maintained by the operating system.

There are two types of SIO FCBs you must set that are required by the SIO procedures:

- File FCB: One file FCB is associated with each opened file, and is passed to each SIO procedure to identify that file:

  --Each file FCB should be declared in the global declaration section of the program.

  --All file FCBs should be initialized prior to calling the OPEN^FILE by using the SET^FILE operation INIT^FILEFCB.

- Common FCB: One common FCB is created for all files. The common FCB contains information common to all files such as a pointer to the error reporting file.

  --The common FCB is usually declared in the global declaration section of the program.

  --The common FCB is automatically initialized during the first call to OPEN^FILE following process creation. OPEN^FILE detects an uninitialized FCB by the fact that its first word (normally the size of the FCB) is zero.

  --If you use the INITIALIZER procedure, the common FCB is declared using define ALLOCATE^CBS and is set to zero automatically.

  --If you do not use the INITIALIZER procedure, you must declare the common FCB and set it to zero before the first call to OPEN^FILE.

  --Do not use the SET^FILE operation INIT^FILEFCB on the common FCB.

You must, as a minimum, put the name of the file to be opened into the FCB by using ASSIGN^FILENAME if you do not use the INITIALIZER procedure and the ALLOCATE^FCB define. Other attributes may have to be set before the file is opened, as shown in the following examples.

Initializing the File FCB Without INITIALIZER

The file FCB must be allocated and initialized before the OPEN^FILE procedure is called to open a file. Use the SET^FILE procedure to do this as explained in the following items.

The following steps illustrate how to initialize the file FCBs without using the INITIALIZER procedure. If you use the INITIALIZER procedure, all of the following are automatically set. When you use the INITIALIZER procedure, all but the first items listed--FCBSIZE, INIT^FILEFCB, and ASSIGN^FILENAME--can be specified.

Dotted underlines show the minimum required syntax; refer to the System Procedure Calls Reference Manual for more information.

1. The size in words of an FCB is provided as a literal,

   FCBSIZE (currently 60)

   For example,

   ```
   INT .INFILE [0:FCBSIZE-1];
   ```

2. Initialize the FCB using the SET^FILE procedure. This step is required:

   ```
   CALL SET^FILE ( <FILE-FCB> , INIT^FILEFCB )
   ---- -------- - ---------- - ------------ -
   ```

   For example,

   ```
   CALL SET^FILE ( INFILE , INIT^FILEFCB )
   ```

3. Specify the name of the file to open. This step is required:

   ```
   CALL SET^FILE (<FILE-FCB>, ASSIGN^FILENAME, <FILENAME-ADDR>)
   ---- -------- ------------ ---------------- ----------------
   ```

   For example,

   ```
   CALL SET^FILE ( INFILE, ASSIGN^FILENAME , @INFILENAME );
   ```

4. Specify the access mode for this open. This step is optional:

   ```
   CALL SET^FILE ( <FILE-FCB>, ASSIGN^OPENACCESS, <OPEN-ACCESS>)
   ---- -------- - ---------- ----------------- -------------
   ```

The following literals are provided for <OPEN-ACCESS> :

```
READWRITE^ACCESS (0)
READ^ACCESS      (1)
WRITE^ACCESS     (2)
```

If omitted, the access mode for the device being opened defaults to the following:

Device	Access
Operator Process	Write
Process	Read-Write
$RECEIVE	Read-Write
Disc	Read-Write
Terminal	Read-Write
Printer	Write
Magnetic Tape	Read-Write
Card Reader	Read

For example,

```
CALL SET^FILE ( INFILE , ASSIGN^OPENACCESS , READ^ACCESS );
```

5. Specify exclusion for this open. This step is optional:

```
CALL SET^FILE ( <FILEFCB> , ASSIGN^OPENEXCLUSION ,
                <OPEN-EXCLUSION> )
```

The following literals are provided for <OPEN-EXCLUSION> :

```
SHARE     (0)
EXCLUSIVE (1)
PROTECTED (3)
```

If omitted, the exclusion mode applied to the open defaults to the following:

Access	Exclusion Mode
Read	If terminal then share, else protected
Write	If terminal then share, else exclusive
Read-Write	If terminal then share, else exclusive

For example,

```
CALL SET^FILE ( INFILE , ASSIGN^OPENEXCLUSION , EXCLUSIVE );
```

6. Specify the logical record length. This step is optional:

```
CALL SET^FILE ( <FILE-FCB> , ASSIGN^RECORDLENGTH ,
                <RECORD-LENGTH> )
```

The <RECORD-LENGTH> is given in bytes.

If omitted, <RECORD-LENGTH> defaults according to the device as follows:

Device	Logical Record Length
Operator Process	132 bytes
Process	132 bytes
$RECEIVE	132 bytes
Unstructured Disc	132 bytes
Structured Disc	Record length defined at creation
Terminal	132 bytes
Printer	132 bytes
Magnetic Tape	132 bytes
Card Reader	132 bytes

7. Set the file code. This step is optional and has two meanings: (1) If AUTO^CREATE is on (default), the file code specifies the type of file to be created (see SIO OPEN^FILE procedure in the System Procedure Calls Reference Manual), and (2) It implies the file code must match the file code specified for the open to succeed:

```
CALL SET^FILE ( <FILE-FCB> , ASSIGN^FILECODE , <FILE-CODE> )
```

8. Set the primary extent size. This step is optional, and has meaning only if AUTO^CREATE is on:

```
CALL SET^FILE ( <FILE-FCB> , ASSIGN^PRIMARYEXTENTSIZE ,
                <PRIMARY-EXTENT-SIZE> )
```

<PRIMARY-EXTENT-SIZE> is given in pages (2048-byte units).

9. Set the secondary extent size. This step is optional, and has meaning only if AUTO^CREATE is on:

```
CALL SET^FILE ( <FILE-FCB> , ASSIGN^SECONDARYEXTENTSIZE ,
                <SECONDARY-EXTENT-SIZE> )
```

<SECONDARY-EXTENT-SIZE> is given in pages (2048-byte units).

10. Set the file's physical block length. This step is optional. The physical block length is the number of bytes transferred between the file and the process in a single I/O operation. If <BLOCK-LENGTH> is specified, blocking is implied. A physical block is composed of <BLOCK-LENGTH> divided by <RECORD-LENGTH> logical records. When <BLOCK-LENGTH> is not exactly divisible by <RECORD-LENGTH>, the portion of that block following the last logical record is filled with blanks.

    Note that the specified form of blocking differs from the type of blocking performed when no <BLOCK-LENGTH> is specified. In the unspecified form, there is no indication of a physical block size; the records are contiguous on the medium:

```
CALL SET^FILE ( <FILE-FCB> , ASSIGN^BLOCKLENGTH , <BLOCK-
LENGTH> )
```

    <BLOCK-LENGTH> is given in bytes.

    If <BLOCK-LENGTH> is not specified, no blocking is performed. Note that <BLOCK-LENGTH> must be specified if files in EDIT format are to be processed.

INTERFACE WITH INITIALIZER AND ASSIGN MESSAGES

The SIO procedures and the INITIALIZER procedure can be used separately or together. It is to your advantage to use the INITIALIZER because it takes care of the entire application startup protocol.

The INITIALIZER procedure provides a way of receiving startup, assign, and param messages without concern for details of the $RECEIVE protocol. The INITIALIZER obtains messages from $RECEIVE and calls the optional user-supplied procedures, passing the messages as parameters to these procedures. Refer to the INITIALIZER procedure in the System Procedure Calls Reference Manual.

The INITIALIZER procedure can prepare global tables of a predefined structure and properly initialize FCBs with the information read from the startup and assign messages. File transfer characteristics such as record length, and even file names, can be altered at run time using by command interpreter ASSIGN commands. Refer to the GUARDIAN Operating System User's Guide.

When using the INITIALIZER with the SIO procedures, you must declare an array called a run-unit control block (RUCB). Each FCB to be prepared by the INITIALIZER must be initialized with a default physical file name and, optionally, with a logical file name before invoking the INITIALIZER.

The INITIALIZER reads the startup message, then requests the assign messages. For each assign message, the FCBs are searched for a logical file name that matches the logical file name contained in the assign message. If a match is found, the information from the assign message is put into the FCB. The assign message is described in Section 5; the ASSIGN command is described in the System Procedure Calls Reference Manual.

The INITIALIZER also substitutes the real file names for default physical file names in the FCBs. Partial file names in the FCBs are expanded using the default volume and subvolume from the startup message. This function provides the capability to define the IN and OUT files of the startup message as physical files and to define the home terminal as a physical file.

After invoking the INITIALIZER, the user program must call the OPEN^FILE procedure once for each file to be opened.

INITIALIZER-RELATED DEFINES

Two defines are provided for allocating run-unit control block space (CBS) and for allocating FCB space. These defines are:

1. Allocate run-unit control block and common FCB (data declaration).

   ALLOCATE^CBS (<rucb> , <common-fcb> , <numfiles>);

   <rucb>

   is the name to be given to the run-unit control block; this name is passed to the INITIALIZER procedure.

   <common-fcb>

   is the name to be given to the common FCB; this name is passed to the OPEN^FILE procedure.

   <numfiles>

   is the number of FCBs to be prepared by the INITIALIZER procedure. The INITIALIZER begins with the first FCB following ALLOCATE^CBS.

   For example,

   ALLOCATE^CBS (RUCB , COMMFCB , 2);

2. Allocate FCB (data declaration).

   NOTE

   The FCB allocation defines must immediately follow the ALLOCATE^CBS define. No intervening variables are allowed.

   ALLOCATE^FCB (<file-fcb> , <default-physical-filename>)

   <file-fcb>

   is the name to be given to the FCB. The name references the file in other sequential I/O procedure calls.

<default-physical-filename>, literal STRING,

is the name of the file to be opened. This can be an internal form of a file name or one of the following, and must be in uppercase as shown:

byte numbers

```
 [0]     [8]     [16]    [24]

"        #IN             "
```

Means substitute the INFILE name of the startup message for this name.

```
"        #OUT            "
```

Means substitute the OUTFILE name of the startup message for this name.

```
"        #TERM           "
```

Means substitute the home terminal name for this name.

```
"        #TEMP           "
```

Means substitute a name appropriate for creating a temporary file for this name.

```
"                        "
```

All blanks means substitute a name appropriate for creating a temporary file for this name.

If the $<volume-name> or <subvolume-name> is omitted, the corresponding default name from the startup message is substituted for the omitted part of the disc file name; for example:

```
ALLOCATE^FCB ( INFILE  , "        #IN             " );
ALLOCATE^FCB ( OUTFILE , "        #OUT            " );
```

The following SET^FILE operation, ASSIGN^LOGICALFILENAME, is used with the INITIALIZER. The logical file name is the means by which the INITIALIZER matches an assign message to a physical file:

```
CALL SET^FILE ( <file-fcb> , ASSIGN^LOGICALFILENAME ,
                @<logical-filename> )
```

<file-fcb>, INT:ref,

references the file to be assigned a logical file name.

@<logical-filename>, INT:value,

is the word address of an array containing the logical file name. A logical file name consists of a maximum of seven letters and digits, the first of which must be a letter.

<logical-filename> must be encoded as follows:

byte numbers

```
[0]  [1]                [8]
<len><logical-filename>
```

<len> is the length, in bytes, of the logical file name.

By convention, the logical file name of the input file of the startup message should be named "INPUT"; the logical file name of the output file of the startup message should be named "OUTPUT"; for example:

```
INT .BUF [0:11];
STRING .SBUF := @BUF '<<' 1;
sbuf ':=' [5, "INPUT"];
CALL SET^FILE (  INFILE , ASSIGN^LOGICALFILENAME , @BUF );
sbuf ':=' [6, "OUTPUT"];
CALL SET^FILE ( OUTFILE , ASSIGN^LOGICALFILENAME , @BUF );
```

Considerations

- If run-time changes to file transfer characteristics using the ASSIGN command are not allowed, then do not assign a logical file name to the file.

- File characteristics can be set by the INITIALIZER, with the ASSIGN command, or with programmatic calls to the SET^FILE procedure. Calls to SET^FILE preceding a call to INITIALIZER are overridden by ASSIGN commands. Calls to SET^FILE following a call to INITIALIZER override ASSIGN commands.

- If you do not want the INITIALIZER to assign a physical file name for the <default-physical-file-name> (for example, you want to restrict all files created by this process to a specific volume, overriding any assign messages), use the following sequence. First declare the FCB:

```
INT .FILE^FCB [0:FCBSIZE - 1];
```

  In the executable part of the program, before calling the INITIALIZER, initialize the FCB:

```
CALL SET^FILE ( FILE^FCB , INIT^FILEFCB );
```

  Assign a logical file name, and any other open attributes desired, before calling the INITIALIZER:

```
CALL SET^FILE ( FILE^FCB, ASSIGN^LOGICALFILENAME, @NAME );

CALL INITIALIZER ( .. );

CALL OPEN^FILE ( COMMON^FCB , FILE^FCB , ... );
```

  If you neglect to ASSIGN a physical file to the logical file, open fails with an error number 513, SIOERR^MISSINGFILENAME, "file name not supplied".

USAGE EXAMPLES

The following example shows the use of the INITIALIZER and SIO procedures for opening the IN and OUT files of a typical Tandem subsystem program. If the IN and OUT files are the same file and either is a terminal or a process, only the IN file is opened for use as both the input and output files. The address of the INFILE FCB is put into the pointer to the OUTFILE FCB so that both pointers refer to the same FCB.

The open access is assigned after the INITIALIZER is called. This overrides the open access specified in an ASSIGN command.

Note that you can assign file names by default, as shown below, or specify them as shown on the next page. If you follow this example, you will have specified file names twice. This is permitted; the last file name specified is used.

Example 1:

```
?NOLIST, SOURCE $SYSTEM.SYSTEM.GPLDEFS
! (The GPLDEFS are listed in Appendix C.)
?LIST
! Set up the control blocks for the INITIALIZER with supplied
! Defines.  Initialize Run Unit Control Block and common FCB.
!     RUCB     - Array holding Run Unit Control Block.
!     COMMFCB  - Array for the common File Control Block.

    ALLOCATE^CBS ( RUCB, COMMFCB, 2 );

! Initialize in file FCB.
!     INFILE   - Array for FCB of the in file.

    ALLOCATE^FCB ( INFILE,  "        #IN             " );

! Initialize out file FCB.
!     OUTFILE - Array for FCB of the out file.

    ALLOCATE^FCB ( OUTFILE, "        #OUT            " );

LITERAL
     process   =    0,              ! Process device type.
     terminal  =    6,              ! Terminal device type.
     inblklen  = 4096,              ! Length of block buffer
                                    ! for in file.
     outblklen = 4096,              ! Length of block buffer
                                    ! for out file.
     rec^len   =  255;              ! Maximum record length
```

```
                                        ! to read or write.

INT .INBLKBUF [0:INBLKLEN/2 - 1],       ! In buffer for blocking.
    .OUTBLKBUF [0:OUTBLKLEN/2 - 1],     ! Out buffer for blocking.
    .INFNAME,                           ! In file's file name.
    .OUTFNAME,                          ! Out file's file name.
     DEVICE^TYPE,                       ! Device type (refer to
                                        ! DEVICEINFO procedure.
     PHYS^REC^LEN,                      ! Physical record length of
                                        ! device.
     INTERACTIVE;                       ! Indicates if in and out
                                        ! files are interactive,
                                        ! implying use of read-write
                                        ! access.
INT .BUF [0:11];                        ! Holds logical file names.
STRING
    .SBUF := @BUF '<<' 1;               ! String corresponding to
                                        ! buffer.
?NOLIST, SOURCE $SYSTEM.SYSTEM.EXTDECS (...)
?LIST

PROC MAIN^PROC MAIN;
  BEGIN
    INT .BUFFER [0: (REC^LEN/2) - 1],   ! Buffer for I/O with a
                                        ! single record.
         COUNT := REC^LEN;              ! Number of bytes read
                                        ! in or written out.
!     Beginning of program execution.

!     Set up in and out files using startup message from RUN
!     command.
        SBUF ':=' [5, "INPUT"];
        CALL SET^FILE( INFILE, ASSIGN^LOGICALFILENAME, @BUF );
        SBUF ':=' [6, "OUTPUT"];
        CALL SET^FILE( OUTFILE, ASSIGN^LOGICALFILENAME, @BUF );
        CALL INITIALIZER( RUCB );

!         get physical file names for in and out files.

         @INFNAME  := CHECK^FILE( INFILE,  FILE^FILENAME^ADDR );
         @OUTFNAME := CHECK^FILE( OUTFILE, FILE^FILENAME^ADDR );

!     Determine type of access for in file.

        CALL DEVICEINFO ( INFNAME, DEVICE^TYPE, PHYS^REC^LEN );
        INTERACTIVE :=
          IF ( DEVICE^TYPE.<4:9> = TERMINAL  OR
               DEVICE^TYPE.<4:9> = PROCESS )
             AND NOT FNAMECOMPARE ( INFNAME, OUTFNAME )
             THEN -1 ELSE 0;
```

```
        CALL SET^FILE( INFILE, ASSIGN^OPENACCESS,
                       IF INTERACTIVE THEN READWRITE^ACCESS
                                      ELSE READ^ACCESS );
!     Open in file, with proper blocking length, sending errors
!     to the out file.

        CALL OPEN^FILE( COMMFCB, INFILE, INBLKBUF
                       ,INBLKLEN,,,,, OUTFILE );

    IF INTERACTIVE THEN       ! Make in and out files the same;
                              !   no need to
      @OUTFILE := @INFILE     !   open out file.
    ELSE                      ! Open out file.
      BEGIN
        CALL SET^FILE(OUTFILE, ASSIGN^OPENACCESS, WRITE^ACCESS);
        CALL OPEN^FILE(COMMFCB, OUTFILE, OUTBLKBUF, OUTBLKLEN );

!         noninteractive use, so echo reads to out file.

        CALL SET^FILE( INFILE, SET^DUPFILE, @OUTFILE );
      END;

!     Main processing loop.

!     WHILE not EOF process the record.

    WHILE ( READ^FILE( INFILE, BUFFER, COUNT) = 0 )  DO
      BEGIN
            .
!           .
!         Process record read in, and format a record for output.
!           .
!           .
        CALL WRITE^FILE( OUTFILE, BUFFER, COUNT );
      END;

   CALL CLOSE^FILE( COMMFCB );    ! close all files

  END;                            ! of MAIN^PROC
```

To change the record length of the input file, you can enter the following ASSIGN command before the program is run:

```
ASSIGN INPUT,,REC 80
```

To change the file code of the output file, you can enter the following ASSIGN command before the program is run:

```
ASSIGN OUTPUT,,CODE 9876
```

Summary

The following are the steps involved to use the INITIALIZER with the SIO procedures:

- Allocate the CBS and FCB, and assign the default physical file names using ALLOCATE^CBS and ALLOCATE^FCBs.
- Assign a logical file name using the SET^FILE operation, ASSIGN^LOGICALFILENAME.
- If ASSIGN command characteristics are to override program calls to SET^FILE, invoke assignment defines.
- Invoke the INITIALIZER to read the startup, assign, and param messages and prepare the file FCBs.
- If programmatic calls to SET^FILE are to override ASSIGN command characteristics, invoke assignment defines.
- Open the files with calls to OPEN^FILE.

Example 2:

The following program example copies an IN file to an OUT file. It also illustrates how to copy an unstructured file into a file in EDIT format and how to read from $RECEIVE into a file in EDIT format.

```
?PAGE "SIODEMO:  Global Space Declaration and Allocation"
NAME SIODEMO;
BLOCK PRIVATE;

DEFINE
  WADDR( x )    = (@x '>>' 1)#,       ! get word addr of str object
  SADDR( x )    = (@x '<<' 1)#,       ! get str addr of word object
  TO^BSZ( x )   = ((x) * 2)#,         ! compute strlen from wordlen
  TO^WSZ( x )   = (((x) + 1) / 2)#,   ! compute wordlen from strlen
  LEN^BSZ( x )  = ($LEN(x))#,         ! get string length of object
  LEN^WSZ( x )  = (TO^WSZ($LEN(x)))#; ! get word length of object

LITERAL
  IOBLK^BSZ     = 4096,               ! I/O buffer block byte size
  IOBLK^WSZ     = TO^WSZ(IOBLK^BSZ), ! I/O buffer block word size
  MSG^BSZ       = 255,                ! max data bytes in a message
  MSG^WSZ       = TO^WSZ(MSG^BSZ);    ! max data words in a message
```

```
!
! STRUCT STARTUP^MSG^DEF -- template for a startup message
!                           (refer to Section 5)
!
STRUCT STARTUP^MSG^DEF(*);
  BEGIN
  INT     MSGID;
  INT     DEFAULT[0:7];
  INT     IN[0:11];
  INT     OUT[0:11];
  STRING  TXT[0:527];
  END;

?NOLIST,SOURCE GPLDEFS    ! (See Appendix C.)
?LIST

!
! Initialize Run Unit Control Block and Common FCB
!   RUCB    -- Array holding Run Unit Control Block
!   CFCB    -- Array holding Common File Control Block
!   3       -- Initialize three FCBs
!
ALLOCATE^CBS(RUCB, CFCB, 3);

! allocate the FCB for the IN file
ALLOCATE^FCB(INFCB,    "        #IN             ");

! allocate the FCB for the OUT file
ALLOCATE^FCB(OUTFCB,   "        #OUT            ");

! allocate the FCB for the ERROR file
ALLOCATE^FCB(ERRORFCB, "        #TERM           ");

INT
 .Inbuf[0:IOBLK^WSZ-1],                  ! block buffer for input
 .Outbuf[0:IOBLK^WSZ-1];                 ! block buffer for output

END BLOCK;
?NOLIST,SOURCE extdecs(
?                      CLOSE^FILE,
?                      INITIALIZER,
?                      OPEN^FILE,
?                      READ^FILE,
?                      SET^FILE,
?                      WRITE^FILE )
?LIST
?PAGE "SIODEMO: PROC startup^message"
!
! PROC STARTUP^MESSAGE--handle startup message provided by the
! INITIALIZER
```

```
!  This routine is presumably called by the INITIALIZER to handle
!  the startup message.  It assumes that the PASSTHRU parameter
!  is really for a buffer big enough to contain a maximum size
!  startup message.  This routine will ensure that there is a
!  trailing null byte in the parameter text field.
!
PROC STARTUP^MESSAGE ( RUCB, PASSTHRU, MESSAGE, MESSLEN, MATCH )
                                                        VARIABLE;
INT    .RUCB;
INT    .PASSTHRU;
INT    .MESSAGE;
INT     MESSLEN;
INT     MATCH;
  BEGIN
  STRING
   .SP;

  PASSTHRU ':=' MESSAGE FOR TO^WSZ( MESSLEN );
  @SP := SADDR( MESSAGE );
  SP[MESSLEN] := 0;          ! null terminate

  END;     ! PROC startup^message

?PAGE "SIODEMO : PROC siodemo MAIN"
!
! PROC SIODEMO -- simple demonstration of SIO usage
!
PROC SIODEMO^30MAY84 MAIN;
  BEGIN
  INT
   .LINEBUF[0:MSG^WSZ-1],              ! input line buffer
    LINECNT;                           ! size of input line
  STRING
   .SLINEBUF := SADDR(LINEBUF);        ! string ptr to buf
  STRUCT
   .STMSG( STARTUP^MSG^DEF );          ! my startup message

  ! get my startup message
  SLINEBUF ':=' [5, "INPUT"];
  CALL SET^FILE(INFCB, ASSIGN^LOGICALFILENAME, @LINEBUF);
  SLINEBUF ':=' [6, "OUTPUT"];
  CALL SET^FILE(OUTFCB, ASSIGN^LOGICALFILENAME, @LINEBUF);
  SLINEBUF ':=' [5, "ERROR"];
  CALL SET^FILE(ERRORFCB, ASSIGN^LOGICALFILENAME, @LINEBUF);
  CALL INITIALIZER(RUCB, STMSG, STARTUP^MESSAGE);

  ! open ERROR !
  CALL SET^FILE(ERRORFCB, ASSIGN^OPENACCESS, WRITE^ACCESS);
  CALL OPEN^FILE(CFCB, ERRORFCB, !blkbuf!, !blkbuflen!,
                                  !flags!, !flagmask!,
                                 !maxreclen!, !prompt!, ERRORFCB);
```

```
  ! open IN !
  CALL SET^FILE(INFCB, ASSIGN^OPENACCESS, READ^ACCESS);
  CALL OPEN^FILE(CFCB, INFCB, INBUF, IOBLK^BSZ);

  ! open OUT
  CALL SET^FILE(OUTFCB, ASSIGN^OPENACCESS, WRITE^ACCESS);
  CALL OPEN^FILE(CFCB, OUTFCB, OUTBUF, IOBLK^BSZ);

  ! process the file
  WHILE ( READ^FILE ( INFCB, LINEBUF, LINECNT, !promptcnt!,
                                                MSG^BSZ) = 0) DO
    BEGIN
    CALL WRITE^FILE(OUTFCB, LINEBUF, LINECNT);
    END;

  ! finish up !
  CALL CLOSE^FILE(CFCB);

  END;     ! PROC SIODEMO MAIN
```

This program example also presents a less obvious method of converting files. Some programmers typically use the following lines to convert a file to EDIT format:

```
: CREATE x
: FUP/ OUT x/ INFO *
: EDIT x  PUT <edit file>
```

Such code requires roughly three times as many disc accesses as the following code, using the above SIODEMO example:

```
: RUN SIODEMO /IN $RECEIVE, OUT<edit file>, NAME $SIO, NOWAIT/
: FUP / OUT $SIO / INFO *
```

Summary

The following operations should be performed in your application code for the SIO functions to work correctly:

1. Call the INITIALIZER to handle the startup message and set up the required FCBs (or use another method to accomplish these tasks; use of the INITIALIZER is recommended).

2. Call OPEN^FILE for each file to be opened for use with the SIO procedures.

3. Call other SIO procedures as required to accomplish the desired tasks.

4. Call CLOSE^FILE when done to close files and flush buffers.

Practice Example

The following example presents a skeleton file copy program using both the INITIALIZER and the SIO procedures. If you want to practice coding variations from a standard skeleton, this example provides a good start.

```
!  Example SIO file copy program--
!  Copies standard input to standard output
!
! Get the SIO literals and defines
?NOLIST, SOURCE $SYSTEM.SYSTEM.GPLDEFS
?LIST

! Define the INITIALIZER's Run Unit Control Block (RUCB) and
! SIO's common FCB and specify that 2 more FCBs will follow

ALLOCATE^CBS ( RUCB, COMMON^FCB, 2 );

! Define FCBs for the IN and OUT files
ALLOCATE^FCB ( IN^FCB,  4 * ["  "] ',' "#IN " ',' 6 * ["  "] );
ALLOCATE^FCB ( OUT^FCB, 4 * ["  "] ',' "#OUT" ',' 6 * ["  "] );

! Get the SIO procedures
?NOLIST, SOURCE $SYSTEM.SYSTEM.EXTDECS ( CLOSE^FILE, INITIALIZER,
?                                        OPEN^FILE,  READ^FILE,
?                                        SET^FILE,   WRITE^FILE )
?LIST

PROC COPY MAIN;
BEGIN

  ! Define the local variables

  ! I/O buffer
  STRUCT .BUFFER;
  BEGIN
    INT i [0:65];
    STRING S [0:131] = i;
  END;

  ! Blocking Buffer definitions

  LITERAL BLOCKING^BUFFER^LENGTH = 1024;    !length in bytes

  STRUCT BLOCKING^BUFFER (*);
  BEGIN
```

```
    INT I [0 : ((BLOCKING^BUFFER^LENGTH + 1) / 2) - 1];
    STRING S [0 : BLOCKING^BUFFER^LENGTH - 1] = i;
  END;

  ! Output blocking buffer
  STRUCT .OUT^BLOCKING^BUFFER ( BLOCKING^BUFFER );
  ! Input blocking buffer
  STRUCT .IN^BLOCKING^BUFFER ( BLOCKING^BUFFER );
  INT EOF,
      LENGTH;

  ! Read the startup, param, and assign messages.  Apply any
  ! assigns for the logical files IN and OUT to IN^FCB and
  ! OUT^FCB.
  CALL INITIALIZER ( RUCB );

  ! Set the IN file for read-only access and the OUT file
  !  for write-only access
  CALL SET^FILE( IN^FCB, ASSIGN^OPENACCESS, READ^ACCESS );
  CALL SET^FILE( OUT^FCB, ASSIGN^OPENACCESS, WRITE^ACCESS );

  ! Open the IN file.  ABEND if open fails.
  CALL OPEN^FILE( COMMON^FCB,
                  IN^FCB,
                  IN^BLOCKING^BUFFER.I,
                  $LEN( IN^BLOCKING^BUFFER )  );

  ! Open the OUT file.  Abend if open fails.
  CALL OPEN^FILE( COMMON^FCB,
                  OUT^FCB,
                  OUT^BLOCKING^BUFFER.I,
                  $LEN( OUT^BLOCKING^BUFFER )  );

  ! Copy loop - Since the defaults are abort on error, disable
  !             break, and ignore system messages, the only
  !             error READ^FILE can encounter is end-of-file on
  !             the input file.

  WHILE NOT READ^FILE( IN^FCB, BUFFER.I, LENGTH ) DO
      CALL WRITE^FILE( OUT^FCB, BUFFER.I, LENGTH );

  ! Close the IN and OUT files.  This must be done to ensure that
  !   any data left in the blocking buffers will be flushed.
  CALL CLOSE^FILE( COMMON^FCB );

END;  ! copy !
```

USAGE EXAMPLE WITHOUT INITIALIZER PROCEDURE

The following example shows the use of the SIO procedures for the IN and OUT files of a typical Tandem subsystem program when the INITIALIZER procedure is not used.

```
?SOURCE $SYSTEM.SYSTEM.GPLDEFS ( ... )
INT  INTERACTIVE,
     ERROR,
    .COMMON^FCB [0:FCBSIZE-1] := 0,
    .RCV^FILE   [0:FCBSIZE-1],
    .INFILE     [0:FCBSIZE-1],
    .OUTFILE    [0:FCBSIZE-1],
    .BUFFER      [0:99],
     MOMPID      [0:3],
     DEVTYPE,
     LENGTH,
     JUNK;

LITERAL
     PROCESS        =     0,
     TERMINAL       =     6,
     IN^BLKBUFLEN   =  1024,
     OUT^BLKBUFLEN  =  1024;

INT .IN^BLKBUF  [0:IN^BLKBUFLEN/2  - 1],
    .OUT^BLKBUF [0:OUT^BLKBUFLEN/2 - 1];

?SOURCE $SYSTEM.SYSTEM.EXTDECS ( ... )
!
! read the startup message.
!
! - open $RECEIVE.
!
CALL SET^FILE ( RCV^FILE , INIT^FILEFCB );
BUFFER ':=' "$RECEIVE  " & BUFFER [4] FOR 7;
! file name.
CALL SET^FILE ( RCV^FILE , ASSIGN^FILENAME , @BUFFER );
! number of bytes to read.
CALL SET^FILE ( RCV^FILE , ASSIGN^RECORDLENGTH , 132 );
CALL OPEN^FILE ( COMMON^FCB , RCV^FILE ,,, NOWAIT , NOWAIT );
!
! - get mom's process ID.
!
! - first, see if I'm named.
!
CALL GETCRTPID ( MYPID , BUFFER );
IF BUFFER.<0:1> = 2 THEN
   ! not named.
   CALL MOM ( MOMPID );
ELSE
```

```
    BEGIN
      ! named.
      CALL LOOKUPPROCESSNAME ( BUFFER );
      MOMPID ':=' BUFFER [5] FOR 4;
    END;
  ! - allow startup message from MOM only.
  CALL SET^FILE ( RCV^FILE , SET^OPENERSPID , @MOMPID );
  !
  DO
    BEGIN
      CALL READ^FILE ( RCV^FILE , BUFFER ,,,, 1 );
      DO ERROR := WAIT^FILE ( RCV^FILE , LENGTH , 3000D )
      UNTIL ERROR <> SIOERR^IORESTARTED;
    END
  UNTIL BUFFER = -1; ! startup message read.

  ! - close $RECEIVE.
  CALL CLOSE^FILE ( RCV^FILE );
  !
  ! see if program is being run interactively.
  !
  CALL DEVICEINFO ( BUFFER [9] , DEVTYPE , JUNK );
  INTERACTIVE :=
     IF ( DEVTYPE.<4:9> = TERMINAL OR
          DEVTYPE.<4:9> = PROCESS  ) AND
          NOT FNAMECOMPARE (BUFFER[9], BUFFER [21] ) THEN 1
                                                 ELSE 0;

  CALL SET^FILE ( INFILE , INIT^FILEFCB );
  CALL SET^FILE ( INFILE , ASSIGN^FILENAME , @BUFFER [9] );
  CALL SET^FILE ( INFILE , ASSIGN^OPENACCESS ,
                  IF INTERACTIVE THEN READWRITE^ACCESS
                                 ELSE READ^ACCESS );
  CALL OPEN^FILE ( COMMON^FCB, INFILE, IN^BLKBUF, IN^BLKBUFLEN
                   ,,,,, OUTFILE );

  IF INTERACTIVE THEN
     ! use in file as out file.
     @OUTFILE := @INFILE
  ELSE
    BEGIN
      CALL SET^FILE (OUTFILE , INIT^FILEFCB );
      CALL SET^FILE (OUTFILE, ASSIGN^FILENAME, @BUFFER[21] );
      CALL SET^FILE (OUTFILE, ASSIGN^OPENACCESS, WRITE^ACCESS);
      CALL OPEN^FILE ( COMMON^FCB , OUTFILE , OUT^BLKBUF ,
                       OUT^BLKBUFLEN );
      ! set duplicative file.
      CALL SET^FILE ( INFILE , SET^DUPFILE , @OUTFILE );
    END;
      .
      .
      .
```

SOURCE FILES

The source file named $SYSTEM.SYSTEM.GPLDEFS must be used with the SIO procedures. It provides the TAL defines and literals for allocating control block space, assigning open characteristics to a file, and for altering and checking file transfer characteristics. The TAL literals for the SIO procedures' error numbers are also included. This file must be referenced in the program's global area before any internal or external procedure declarations or within a procedure before any subprocedure declarations.

SIO Considerations

Trying to do I/O to a file that is not open results in ABEND.

If a file is not open, SIO guarantees that the file number in the initialized FCB is -1. This can be verified by using the CHECK^FILE operation FILE^FNUM^ADDR.

All errors print only to the last error file specified. (Default is the home terminal.)

SIO takes advantage of the assign message mechanism to set up logical file names; file names of up to 7 characters are allowed.

SIO allows read or write access for files in EDIT format and access for blocked files--read-write access is not permitted.

The SIO blocking buffers you specify for files in EDIT format must be of sufficient size. If the block size specified is too small, error 518, SIOERR^BUFTOOSMALL, is returned. Minimum buffer block and buffer size requirements are noted under the OPEN^FILE procedure later in this section. The buffer you pass to OPEN^FILE for use with files in EDIT format must be large enough to contain the compressed text images and the page frame directory for the EDIT file.

SIO has the capability to wait for a line printer or other device that is not ready. This can be important, for example, if you run a system without a Spooler. (SIO reacts, as does FUP, with a notice to the home terminal when a device is not ready.)

SIO can write to all device types correctly--including writing to tape and appending the appropriate EOF marks. SIO CLOSE^FILE closes all files correctly and flushes the associated buffers. Failure to call CLOSE^FILE can lead to corrupted files.

It is easier to rely on the BREAK handling capabilities of the SIO procedures than to do your own BREAK handling; in any case, do not mix your own BREAK handling procedures with those of SIO.

If a checksum error occurs, you should assume that your FCB is ruined. This is usually caused by coding errors in your program. You should carefully check your program before starting over.

If you want to write to a file in EDIT format, use the following specifications:

```
CALL SET^FILE( OUTFILE, ASSIGN^OPENACCESS, WRITE^ACCESS );
CALL OPEN^FILE; !with block buffer length of 1024 or greater
```

If you want to read a file in EDIT format, use:

```
CALL SET^FILE( INFILE, ASSIGN^OPENACCESS, READ^ACCESS );
CALL OPEN^FILE; !with block buffer length of 144 or greater
```

For increased performance with files in EDIT format, use larger block sizes. If error 518 (SIOERR^BUFTOOSMALL) occurs, it is recommended that you double the previously used buffer size before proceeding. Do not specify more than 16K bytes (including the directory).

$RECEIVE HANDLING

Within the environment of the SIO procedures, the $RECEIVE file has two functions:

- To check for break messages
- To transfer data between processes

Within the SIO procedures, these functions can be performed concurrently. It may be desirable to manage the $RECEIVE file independently of the SIO procedures, and to monitor BREAK using the SIO procedures. Therefore, the SET^FILE operation SET^BREAKHIT enables the user's $RECEIVE handler to pass the BREAK information into the SIO procedure environment.

The interaction between SIO procedures, $RECEIVE, and BREAK is limited by the following considerations:

- SIO procedures assume that if you have opened $RECEIVE, you will read from it. BREAK messages are ignored if you do not read from $RECEIVE after opening it.
- Although GUARDIAN allows a process to own BREAK on an arbitrary number of terminals, SIO supports BREAK ownership for only one terminal at a time.
- SIO does not support BREAK access. In other words, it always issues SETMODE 11 with parameter 2 = 0. SETMODE 11 is described at the end of Section 6; the BREAK feature is also discussed in Section 6.
- If a process launches an offspring process that takes BREAK ownership, and the parent then calls CHECK^BREAK, SIO will take BREAK ownership back.

For more information on $RECEIVE, refer to Section 4.

NOWAIT I/O

If NOWAIT is specified at open time, the file is opened with a nowait I/O depth of one. Whether an individual operation is to be waited for is determined on a call-by-call basis. Nowait operations are completed by a call to WAIT^FILE.

If it is desirable to wait for any file (as opposed to a particular file), you can call AWAITIO before calling WAIT^FILE. Depending on whether blocking is performed, a physical I/O operation may not always take place with a logical I/O operation. You can use the CHECK^FILE operation FILE^PHYSIOOUT to determine if a physical I/O operation is outstanding. The SET^FILE operations SET^PHYSIOOUT, SET^ERROR, and SET^COUNTXFERRED are provided to modify the FCB if the I/O is completed. The user must call WAIT^FILE following the call to AWAITIO for the file state information to be updated; for example,

```
INT .IN^FNUM;

@IN^FNUM := CHECK^FILE ( INFILE , FILE^FNUM^ADDR );
ERROR := 0;
WHILE 1 DO
  BEGIN
    IF ERROR <> SIOERR^IORESTARTED THEN
       CALL READ^FILE ( INFILE , BUFFER , , , 1 ); ! no wait.
    .
    .
    FNUM := -1;
    CALL AWAITIO ( FNUM ,, COUNTREAD ,, 3000D );
    IF FNUM = IN^FNUM THEN
      BEGIN
        CALL FILEINFO ( IN^FNUM , ERROR );
        ! set I/O done.
        CALL SET^FILE ( INFILE , SET^PHYSIOOUT , 0 );
        ! set count read.
        CALL SET^FILE (INFILE , SET^COUNTXFERRED, COUNTREAD);
        ! set error code.
        CALL SET^FILE ( INFILE , SET^ERROR , ERROR );
        IF ( ERROR :=
           WAIT^FILE ( INFILE , INFILE^COUNTREAD ) ) <>
              SIOERR^IORESTARTED THEN
          BEGIN ! completed.
            !
            ! process read.
            !
          END;
      END
    ELSE
      .
      .
```

SECTION 18

FORMATTER

The GUARDIAN operating system formatter provides you with the capability to format data while it is output and to convert data already input with a minimum of programming effort. The formatter consists of two procedures that can be called from user programs.

The formatter procedures are:

FORMATCONVERT converts an external format to internal form for presentation to the FORMATDATA procedure.

FORMATDATA performs conversion between internal and external representation of data as specified by a format, or performs conversion of data using the list-directed rules.

These two procedures are fully described in the System Procedure Calls Reference Manual. You will need to refer to that manual to use the parameters illustrated in the examples presented here.

NOTE

The decimal arithmetic package is required to use the formatter.

The floating-point arithmetic package is needed when using the D, E, and G edit descriptors for output, or when floating-point variables are used.

FORMAT-DIRECTED FORMATTING

The principal parameters to the formatter are a list of data elements, an array of buffers, and a format.

The format is a list of edit descriptors, separated by commas, that are translated into internal form by FORMATCONVERT for presentation to FORMATDATA. Edit descriptors may optionally be preceded by one or more modifiers and/or decorations, enclosed in brackets ([]), specifying additional field formatting. The FORMATCONVERT procedure converts the external data into an internal form for presentation to the FORMATDATA procedure.

The FORMATDATA procedure matches each data element with its associated edit descriptor, which specifies how it is to be displayed for output or how the buffer contents are to be interpreted for input. FORMATDATA proceeds through the list of edit descriptors, from left to right, in the order in which they were presented. If an edit descriptor is a nonrepeatable item, FORMATDATA processes it directly; if an edit descriptor is a repeatable item, FORMATDATA obtains the next data element from the data list and performs the data conversion specified by the edit descriptor. This processing continues until the data list is exhausted.

Exceptions to the left-to-right processing are the repeat factor and format loopback. Any edit descriptor, or groups of edit descriptors enclosed in parentheses, can be applied repeatedly to a number of data values by a positive-integer repeat factor preceding the descriptor or group. If the end of the format is reached with unprocessed data elements remaining, format loopback selects the portion of the format to be interpreted again.

The <variablelist> defines a sequence of variables or arrays that are to be processed by the FORMATDATA procedure. Each variable or element of an array in the <variablelist> is referred to as a data element.

Format Characteristics

A format directs the operation of the FORMATDATA procedure editing between the internal and external representation of data.

The form of a format is:

```
                { fmt-item    }   [ separator   fmt-item    ]
format:         {             }   [                         ] ...
                { b-separator }   [ [separator] b-separator ]

                { nonrepeatable-edit-descriptor  }
fmt-item:       {                                }
                {         field-group            }

field-group:  [repeat]  [mods]  { group-spec }

                { repeatable-edit-descriptor }
group-spec:     {                            }
                {     "(" format  ")"        }

repeat:       an unsigned, nonzero integer

                      { modifier  }
mods:           "["   {           } , . . .  "]"
                      { decoration }

separator:    { , | / | : }

b-separator:  { / | : }
```

nonrepeatable edit descriptors		repeatable edit descriptors		modifiers		decorations	
BN	SP	A	G	BN	OC	{ P }	
BZ	SS	D	I	BZ	RJ	{ M }	{ A }
H	T	E	L	FL	SS	{ Z }	{ F }
string	TL	F	M	LJ		{ N }	{ P }
P	TR					{ O }	
S	X						

Some sample formats:

```
I5,F10.2

"        NAME          EXTENSION",//,(A20,3X,I4)

3("   ITEM    ACTIVITY",10X),//,3(M<99-9999>,
                                  5X,[BZ]I6,1X,10X)
```

FORMATDATA matches each data element with its associated edit descriptor, which specifies how it is to be displayed for output or how the buffer contents are to be interpreted for input. FORMATDATA proceeds through your list (from left to right) of edit descriptors in the order in which they are presented:

1. If an edit descriptor is a repeatable item, FORMATDATA obtains the next data element from the data list and performs the data conversion specified by the edit descriptor.

2. If an edit descriptor is a nonrepeatable item, FORMATDATA processes it directly.

This processing continues until the data list is exhausted. If there are any data list items, you must include at least one repeatable edit descriptor in the format.

The interpretation of the format terminates if any of these conditions are met:

1. FORMATDATA encounters a repeatable edit descriptor in the format and there are no remaining data elements.

2. FORMATDATA reaches the end of the format and there are no remaining data elements.

3. FORMATDATA encounters a colon edit descriptor in the format and there are no remaining data elements.

A format is interpreted from left to right with the following exceptions:

1. If a field group contains a repeat factor, then the group specifications are processed the number of times indicated by the repeat factor before continuing with the following specifications.

2. If FORMATDATA reaches the end of the format and data elements remain, format loopback occurs. Format loopback performs the following steps:

   --The current buffer is terminated.

   --A new buffer is obtained.

   --The format is examined backwards (from right to left). If a right parenthesis that is not part of a string or Hollerith descriptor is encountered, the matching left parenthesis is found, and the format interpretation resumes at the left parenthesis. If a repeat factor precedes this left parenthesis,

processing resumes at the repeat factor. If the beginning of the format is reached and no right parenthesis is found, the format interpretation resumes at the beginning.

--Reverse examination of the format position has no effect on the scale factor (set by P), the sign control (set by S, SP, or SS), or the blank control (set by BN or BZ). The condition in effect at the end of the format continues until altered by one of the controlling edit descriptors.

Example

This example shows how to use the formatter procedures for some simple output editing. It illustrates the setup for the variable list and the use of the <length> values returned from FORMATDATA.

```
PROC EXAMPLE MAIN;
   BEGIN
!
! This structure defines the four-word form of variable list
! entry (the one without the null value pointer field).
!
   STRUCT VLE^REF (*);
      BEGIN
      INT    ELE^PTR ;
      STRING ELE^SCALE, ELE^TYPE ;
      INT    ELE^LEN, ELE^OCCURS ;
      END ;
!
! This DEFINE provides one way to initialize the fields of a
! variable list entry.  The SCALE, TYPE, LENGTH, and OCCURS
! values must be constants to use it.
!
   DEFINE VLE^INIT (ENT, V, SCALE, TYPE, LEN, OCCURS) =
               BEGIN
               ENT ':=' [0,SCALE '<<' 8 '+' TYPE, LEN, OCCURS] ;
               ENT.ELE^PTR := @V ;
               END  #;
!
! This structure defines a buffer to make it easier to create
! an array of buffers.
!
   LITERAL BUF^LEN = 100 ;
   STRUCT BUF^REF (*) ;
      BEGIN
      STRING BYTES [0:BUF^LEN-1] ;
      END ;
```

```
!
! The example format in external (ASCII) form.
!
   LITERAL EFORMATLEN = 60 ;
   STRING .EFORMAT [0:EFORMATLEN] :=
             "20X,'SAMPLE OUTPUT' // I5,2X,F10.3,5(2X,I2),5X,A" ;
!
! Storage for the internal form of the format.
!
   LITERAL IFORMATLEN = 200 ;
   INT    .WFORMAT [0:IFORMATLEN/2] ;
   STRING .IFORMAT := @WFORMAT '<<' 1 ;
!
! Array of buffers and the length used in each.
!
   LITERAL NUM^BUFS = 5 ;
   STRUCT .BUFFERS ( BUF^REF ) [0:NUM^BUFS-1] ;
   INT    .BUF^LENS [0:NUM^BUFS-1] ;
!
! Variable list array.
!
   STRUCT .VLIST ( VLE^REF ) [0:3] ;
!
! Data for the example.
!
   INT      INT^16 :=  7 ;
   FIXED(2) QUAD := -437.57F ;
   INT(32) .INT^32^ARRAY [0:4] := [1D, 1D, 2D, 3D, 5D] ;
   STRING  .CHARS [0:10] := "DEMO STRING" ;
!
! Miscellaneous data.
!
   INT     .FILENAME [0:11] ;
   INT      FILENO ;
   INT      SCALES, ERROR, I ;
!
! Initialization.
!
   CALL MYTERM ( FILENAME ) ;
   CALL OPEN ( FILENAME, FILENO ) ;
!
! Convert the format to internal form.
! Note the way to ignore the SCALE information.
!
   SCALES := 0 ;
   ERROR := FORMATCONVERT ( IFORMAT, IFORMATLEN, EFORMAT,
                            EFORMATLEN, SCALES, SCALES, 1 ) ;
   IF ERROR <= 0 THEN BEGIN
                        !  here if error in FORMAT
                      END ;
```

```
!
! Set up the variable list entries, both by using the DEFINE
! and by separate stores into the ITEM fields.
!
   VLE^INIT ( VLIST[0], INT^16, 0, 2, 2, 1 ) ;
                ! SCALE 0, TYPE 2, LEN 2 BYTES, 1 OCCURRENCE
   VLE^INIT ( VLIST[1], QUAD, 2, 6, 8, 1 ) ;
                ! SCALE 2, TYPE 6, LEN 8 BYTES, 1 OCCURRENCE
   VLIST[2].ELE^PTR    := @INT^32^ARRAY ;      ! VARIABLE ADDRESS
   VLIST[2].ELE^SCALE  := 0 ;                  ! SCALE 0
   VLIST[2].ELE^TYPE   := 4 ;                  ! TYPE 4
   VLIST[2].ELE^LEN    := 4 ;                  ! LENGTH 4 BYTES
   VLIST[2].ELE^OCCURS := 5 ;                  ! 5 OCCURRENCES

   VLE^INIT ( VLIST[3], CHARS, 0, 0, 11, 1 ) ;
                ! SCALE 0, TYPE 0, LEN 11 BYTES, 1 OCCURRENCE
!
! Edit the data into the buffers.
!
   ERROR := FORMATDATA ( BUFFERS, BUF^LEN, NUM^BUFS, BUF^LENS,
                         WFORMAT, VLIST, 4, 0 ) ;
   IF ERROR <> 0 THEN BEGIN
                         ! here if error in data conversion
                      END ;
!
! Write the buffers used to the terminal.
!
   I := 0 ;
   WHILE  I <= NUM^BUFS  AND  BUF^LENS[I] >= 0  DO  BEGIN
              CALL WRITE ( FILENO, BUFFERS[I], BUF^LENS[I] ) ;
              I := I + 1 ;
              END ;
!
! The output produced is the three lines shown below.
! The "|" character is used to show the buffer limits indicated
! in the BUF^LENS array:
!
!   |                       SAMPLE OUTPUT|
!   ||
!   |    7    -437.570   1   1   2   3   5     DEMO STRING|
!
   CALL STOP ;

   END ;   !  EXAMPLE
```

EDIT DESCRIPTORS

Edit descriptors are of two types: those that specify the conversion of data values (repeatable) and those that do not (nonrepeatable). The effect of repeatable edit descriptors can be altered through the use of modifiers or decorations, which are enclosed in brackets ([]) preceding the edit descriptors to which they refer. Within a format, all edit descriptors except buffer control descriptors must be separated by commas. Buffer control descriptors have the dual function of edit descriptors and format separators, and need not be set off by commas.

All the descriptors, modifiers, and decorations are summarized here and fully explained following this summary.

Summary of Nonrepeatable Edit Descriptors

The edit descriptors that are not associated with data items are of six subtypes:

- Tabulation
  1. Tn --Tab absolute to nth character position
  2. TRn --Tab right
  3. TLn --Tab left
  4. nX --Tab right (same as TR)
- Literals
  1. Alphanumeric string enclosed in apostrophes (') or quotation marks (")
  2. Hollerith descriptor (nH followed by n characters)
- Scale factor specification
  1. P --Implied decimal point in a number

- Optional plus control

  These descriptors provide control of the appearance of an optional plus sign for output formatting. They have no effect on input.

  1. S --Do not supply a plus
  2. SP --Supply a plus
  3. SS --Do not supply a plus

- Blank interpretation control

  1. BN --Blanks ignored (unless entire field is blank)
  2. BZ --Blanks treated as zeros

- Buffer control

  1. / --Terminate the current buffer, and then obtain a new one
  2. : --Terminate formatting if no data elements remain

Summary of Repeatable Edit Descriptors

Repeatable edit descriptors direct the formatter to obtain the next data list element and perform a conversion between internal and external representation. They may be preceded by modifiers or decorations that alter the interpretation of the basic edit descriptor. Modifiers and decorations apply only to output conversion. They are allowed but ignored for input.

The repeatable edit descriptors are:

1. A --Alphanumeric (ASCII)
2. D,E --Exponential form
3. F --Fixed form
4. G --General (E or F format depending on magnitude of data)
5. I --Integer
6. L --Logical
7. M --Mask formatting

Summary of Modifiers

Modifiers are codes that are used to alter the results of the formatting prescribed by the edit descriptors to which they are attached. They are:

1. BN, BZ --Field blanking (if null, or zero)
2. FL --Fill-character specification
3. LJ, RJ --Left and right justification
4. OC --Overflow-character modifier
5. SS --Symbol substitution

Summary of Decorations

Decorations specify alphanumeric strings that can be added to a field either before basic formatting is begun or after it is finished. A decoration consists of one or more codes that specify the conditions under which the string is to be added (based on the value of the data element or the occurrence overflow of the external field):

1. M --Minus
2. N --Null
3. O --Overflow
4. P --Plus
5. Z --Zero

followed by a code that describes the position of the special editing:

1. A (absolute) --at a specific character position within the field
2. F (floating) --at the position the basic formatting finished
3. P (prior) --at the position the basic formatting would have started

followed by the character string that is to be included in the field if the stated conditions are met.

NONREPEATABLE EDIT DESCRIPTORS

The following descriptions show the form, function, and requirements for each of the nonrepeatable edit descriptors.

Tabulation Descriptors

The tabulation descriptors specify the position at which the next character is transmitted to or from the buffer. This allows portions of a buffer to be processed in an order other than strictly left to right, and permits processing of the same portion of a buffer more than once.

The forms of the tabulation descriptors are as follows (n is an unsigned integer constant):

Tn	TLn	TRn	nX

- Tn --indicates that the transmission of the next character to or from a buffer is to occur at the nth character position. The first character of the buffer is numbered 1.
- TLn --indicates that the transmission of the next character to or from the buffer is to occur n positions to the left of the current position.
- TRn --indicates that the transmission of the next character to or from the buffer is to occur n positions to the right of the current position.
- nX --is exactly identical to TRn above.

Each of these edit descriptors alters the current position but has no other effect.

The current position may be moved beyond the limits of the current buffer (that is, become less than or equal to zero, or greater than <bufferlen>) without an error resulting, provided that no attempt is made by a subsequent edit descriptor to transmit data to or from a position outside the current buffer.

Tab descriptors may not be used to advance to later buffers or to return to previous ones. The following examples illustrate tabulation descriptors:

Data List Values

```
100
1000.49F
"HELLO"
```

	Format	Results
No tabs	I3,E12.4,A5	100 0.1000E+04HELLO
X	I3,E12.4,1X,A5	100 0.1000E+04 HELLO
TL	I3,E12.4,TL3,A5	100 0.1000EHELLO
TR	I3,E12.4,TR5,A5	100 0.1000E+04 HELLO
T	I3,E12.4,T3,A5	10HELLO1000E+04

The "/\" marker denotes the boundaries of the output field.

Literal Descriptors

Literal descriptors are alphanumeric strings in either form:

```
dc c c  ... c d     OR     nHc c c  ... c
  1 2 3      n                1 2 3      n
```

d = either an apostrophe (') or a quotation mark (");
the same character must be used for both the opening and closing delimiters.

c = any ASCII character.

n = an unsigned nonzero integer constant specifying the number of characters in the string; n cannot exceed 255.

On input, a literal descriptor is treated as nX.

A literal edit descriptor causes the specified character string to be inserted in the current buffer beginning at the current position. It advances the current position n characters.

In a quoted literal form, if the character string to be represented contains the same character that is used as the delimiter, two consecutive characters are used to distinguish the data character from the delimiter; for example,

To represent:	Use:
can't	'can''t' or "can't"
"can't"	'"can''t"' or """can't"""

In the Hollerith constant form, the number of characters in the string (including blanks) must be exactly equal to the number preceding the letter H. There are no delimiter characters, so the characters are supplied exactly as they should appear in the buffer; for example,

To represent:	Use:
can't	5Hcan't

Scale-Factor Descriptor (P)

The form of a scale-factor descriptor is:

```
nP

n = optionally signed integer in the range of -128 to 127.
```

The value of the scale factor is zero at the beginning of execution of the FORMATDATA procedure. Any scale-factor specification remains in effect until a subsequent scale specification is processed. The scale factor applies to the D, E, F, and G edit descriptors, affecting them in the following manner:

1. On input, with D, E, F, and G edit descriptors (provided no exponent exists in the external field), the scale-factor effect is that the externally represented number equals the internally represented number multiplied by 10**n.

2. On input, with D, E, F, and G edit descriptors, the scale factor has no effect if there is an exponent in the external field.

3. On output, with D and E edit descriptors, the mantissa of the quantity to be produced is multiplied by 10**n, and the exponent is reduced by n.

4. On output, with the F edit descriptor, the scale-factor effect is that the externally represented number equals the internally represented number multiplied by 10**n.

5. On output, with the G edit descriptor, the effect of the scale factor is suspended unless the magnitude of the data to be processed is outside the range that permits the use of an F edit descriptor. If the use of the E edit descriptor is required, the scale factor has the same effect as with the E output processing.

Optional Plus Descriptors (S, SP, SS)

Optional plus descriptors may be used to control whether optional plus characters appear in numeric output fields. In the absence of explicit control, the formatter does not produce any optional plus characters.

The forms of the optional plus descriptors are:

S	SP	SS

These descriptors have no effect upon input.

If the S descriptor is encountered in the format, the formatter does not produce a plus in any subsequent position that normally contains an optional plus.

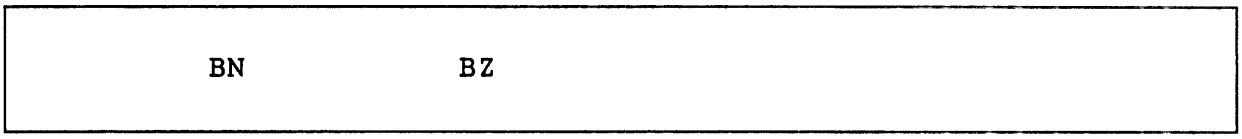
If the SP descriptor is encountered in the format, the formatter produces a plus in any subsequent position that normally contains an optional plus.

The SS descriptor is the same as S (above).

An optional plus is any plus except those appearing in an exponent.

Blank Descriptors (BN, BZ)

The blank descriptors have the following form:



These descriptors have no effect on output.

The BN and BZ descriptors may be used to specify the interpretation of blanks, other than leading blanks, in numeric input fields. At the beginning of execution of the FORMATDATA procedure, nonleading blank characters are ignored.

If a BZ descriptor is encountered in a format, all nonleading blank characters in succeeding numeric input fields are treated as zeros.

If a BN descriptor is encountered in a format, all blank characters in succeeding numeric input fields are ignored. The effect of ignoring blanks is to treat the input field as if all blanks had been removed, the remaining portion of the field right-justified, and the blanks reinserted as leading blanks. However, a field of all blanks has the value zero.

The BN and BZ descriptors affect the D, E, F, G, and I edit descriptors only.

Buffer Control Descriptors (/, :)

There are two edit descriptors used for buffer control:

/	indicates the end of data list item transfer on the current buffer and obtains the next buffer. The current position is moved to 1 in preparation for processing the next buffer.
:	indicates termination of the formatting provided there are no remaining data elements.

- To clarify, the operation of the slash (/) is as follows for any positive integer n:

  --If n consecutive slashes appear at the end of a format, this causes n buffers to be skipped.

  --If n consecutive slashes appear within the format, this causes n-1 buffers to be skipped.

- The colon (:) is used to conditionally terminate the formatting. If there are additional data list items, the colon has no effect. The colon can be of use when data items are preceded by labels, as in the following example:

```
10(' NUMBER ',I1,:/)
```

This group of edit descriptors is preceded by a repeat factor that specifies the formatting of ten data items, each one to be preceded by the label NUMBER. If there are fewer than ten data items in the data list, formatting terminates immediately after the last value is processed. If the colon is not present, formatting continues until the I edit descriptor is encountered for the fourth time. This means the fourth label is added before the formatting is terminated.

The following example illustrates this usage:

Data Items:

```
1
2
3
```

Format:

With colon	Without colon
`10('NUMBER ',I1,:/)`	`10('NUMBER ',I1,/)`

Results:

```
|NUMBER 1|
|NUMBER 2|
|NUMBER 3|
```

```
|NUMBER 1|
|NUMBER 2|
|NUMBER 3|
|NUMBER |
```

The "|" character is used to denote the boundaries of the output field.

REPEATABLE EDIT DESCRIPTORS

The following descriptions give the form, function, and requirements for each of the edit descriptors that specify formatting of data fields. The following edit descriptors can be preceded by an unsigned integer repeat factor to specify identical formatting for a number of values in the data list.

The following descriptions of the operation of repeatable edit descriptors apply when no decorations or modifiers are present.

The A Edit Descriptor

The A edit descriptor is used to move characters between the buffer and the data element without conversion. This is normally used with ASCII data.

The A edit descriptor has one of the following forms:

Aw	OR	A

w = an unsigned integer constant that specifies the width, in characters, of the field and may not exceed 255. The field processed is the next w characters starting at the current position.

If w is not present, the field width is equal to the actual number of bytes in the associated data element, but cannot exceed 255. Values over 255 are reduced to 255.

After the field is processed, the current position is advanced by w characters.

On output, the operation of the A edit descriptor is as follows:

1. The number of characters specified by w, or the number of characters in the data element, whichever is less, is moved to the external field. The transfer starts at the left character of both the data element and the external field unless an RJ modifier is affecting the descriptor, in which case the transferring of characters begins with the right character of each.

2. If w is less than the number of characters in the data element, the field overflow condition is set.

3. If w is greater than the number of characters in the data element, the remaining characters in the external field are filled with spaces (unless another fill character is specified by the FL modifier).

It is not mandatory that the data element be of type character. For example, an INTEGER(16) element containing the octal value %015536 corresponds to the ASCII characters "ESC" and "^", which can be output to an ADM-2 terminal using an A2 descriptor to control a blinking field on the screen; for example,

Format	Data Value	External Field	
A	'WORD'	|WORD|	
A4	'WORD'	|WORD|	
A3	'WORD'	|WOR|	(overflow set)
[RJ]A3	'WORD'	|ORD|	(overflow set)
A5	'WORD'	|WORD |	
[RJ]A5	'WORD'	| WORD|	
A	%044111	|HI|	

In the last example, the data value was stored in a 2-byte INTEGER.

The "|" character is used to denote the boundaries of the output field.

On input, the operation of the Aw edit descriptor is as follows:

1. The number of characters specified by w, or the number of characters contained in the data element specified by n, whichever is less, is moved from the external field to the data element. The transfer begins at the left character of both the data element and the external field.

2. If w is less n, the data element is filled with (n-w) blanks on the right.

3. If w is greater than n, the leftmost n characters of the field are stored in the data element.

The following examples illustrate these considerations:

External Field	Format	Data Item Length	Data Element Value
|HELLO|	A5	5 characters	'HELLO'
HELLO|	A3	3 characters	'HEL'
HELLO |	A6	6 characters	'HELLO '
|HELLO|	A5	6 characters	'HELLO '
|HELLO|	A5	3 characters	'HEL'

The "|" character is used to denote the boundaries of the input field.

The D Edit Descriptor

The exponential edit descriptor is used to display or interpret data in floating-point form, usually used when data values have extremely large or extremely small magnitude.

The D edit descriptor is of the form:

Dw.d

This descriptor is identical to the Ew.d descriptor.

This edit descriptor is used in the same manner as the E edit descriptor (below).

NOTE

To use the D edit descriptor for output, floating-point firmware is required.

The E Edit Descriptor

The exponential edit descriptor is used to display or interpret data in floating-point form. It is usually used when data values have extremely large or extremely small magnitude.

The E edit descriptor has one of the following forms:

Ew.d OR Ew.dEe

w = an unsigned integer constant that defines the total field width (including the exponent) and can not exceed 255. The field processed is the w characters starting at the current position. After the field is processed, the current position is advanced by w characters.

d = an unsigned integer constant that defines the number of digits that are to appear to the right of the decimal point in the external field.

e = an unsigned integer constant that defines the number of digits in the exponent. If Ew.d is used, e takes the value 2.

The input field consists of an optional sign, followed by a string of digits optionally containing a decimal point. A decimal point appearing in the input field overrides the portion of the descriptor that specifies the decimal point location. However,if you omit the decimal point, the rightmost d digits of the string, with leading zeros assumed if necessary, are interpreted as the fractional part of the value represented. The string of digits may be of any length. Those beyond the limit of precision of the internal representation are ignored. The basic form may be followed by an exponent in one of the following forms:

1. Signed integer constant.

2. E followed by zero or more blanks, followed by an optionally signed integer constant.

3. D followed by zero or more blanks, followed by an optionally signed integer constant.

An exponent containing a D is processed identically to an exponent containing an E.

On output, the field (for a scale factor of zero) appears in the following form:

```
{[+]} [0].n n  ... n      E {+}  e e ...e
{ - }      1 2     d        {-}   1 2    e
```

{[+]} { - }	indicates an optional plus or a minus.
n n ... n 1 2 d	are the d most significant digits of the value of the data after rounding.
E	signals the start of the decimal exponent.
{+} {-}	indicates that a plus or minus is required.
e e ... e 1 2 e	are the e most significant digits of the exponent.

The sign in the exponent is always displayed. If the exponent is zero, a plus sign is used.

If the data is negative, the minus sign is always displayed. If the data is positive (or zero), the display of the plus sign is dependent on the last optional plus descriptor processed.

The zero preceding the decimal point is normally displayed, but may be omitted to prevent field overflow.

Decimal normalization is controlled by the scale factor established by the most recently interpreted nP edit descriptor. If $-d < n \leq 0$, the output value has $|n|$ leading zeros, and $(d-|n|)$ significant digits follow the decimal point; if $0 < n < d+2$, the output value has n significant digits to the left of the decimal point and d-n+1 digits to the right. If the number of characters produced exceeds the field width or if an exponent exceeds its specified length using the Ew.dEe field descriptor, the entire field of width w is filled with asterisks. However, if the field width is not exceeded when optional characters are omitted, the field is displayed without the optional characters.

Because all characters in the output field are included in the field width, w must be large enough to accommodate the exponent, the decimal point, and all digits and the algebraic sign of the base number.

The following examples illustrate output:

Format	Data Value	Result
E12.3	8.76543 x 10	0.877E-05
E12.3	-0.55555	-0.556E+00
E12.3	123.4567	0.123E+03
E12.6E1	3.14159	0.314159E+1

The "|" character is used to denote the boundaries of the output field.

NOTE

To use the E edit descriptor for output, floating-point firmware is required.

The following examples illustrate input:

External Field	Format	Data Element Value
0.100E+03	E12.3	100
100.05	E12.5	100.05
12345	E12.3	12.345

The "|" character is used to denote the boundaries of the output field.

The F Edit Descriptor

The fixed-format edit descriptor is used to display or interpret data in fixed point form.

The F edit descriptor has the following forms:

Fw.d OR Fw.d.m

w = an unsigned integer constant that defines the total field width and cannot exceed 255. The field processed is the w characters starting at the current position. After the field is processed, the current position is advanced by w characters.

→

d = an unsigned integer constant that defines the number of digits that are to appear to the right of the decimal point in the external field.

m = an unsigned integer constant that defines the number of digits that must be present to the left of the decimal point on output.

On input, the Fw.d edit descriptor is the same as the Ew.d edit descriptor.

The output field consists of blanks if necessary, followed by a minus if the internal value is negative or an optional plus otherwise. This is followed by a string of digits that contains a decimal point and represents the magnitude of the internal value, as modified by the established scale factor and rounded to the d fractional digits. If the magnitude of the value in the output field is less than one, there are no leading zeros except for an optional zero immediately to the left of the decimal point. The optional zero must appear if there would otherwise be no digits in the output field. If the Fw.d.m form is used, leading zeros are supplied if needed to satisfy the requirement of m digits to the left of the decimal point; for example,

Format	Data Value	Result
F10.4	123.4567	123.4567|
F10.4	0.000123	0.0001|
F10.4.3	-4.56789	-004.5679|

The "|" character is used to denote the boundaries of the output field.

The G Edit Descriptor

The general format edit descriptor can be used in place of either the E or the F edit descriptor, since it has a combination of the capabilities of both.

The G edit descriptor has either of the forms:

```
Gw.d          OR          Gw.dEe

w = an unsigned integer constant that defines the total
    field width and may not exceed 255.  The field
    processed is the w characters starting at the
    current position.  After the field is processed,
    the current position is advanced by w characters.

d = an unsigned integer constant that defines the number
    of significant digits that are to appear in the
    external field.

e = an unsigned integer constant that defines the number
    of digits in the exponent, if one is present.
```

On input, the G edit descriptor is the same as the E edit descriptor.

The method of representation in the output field depends on the magnitude of the data being processed, as follows:

Magnitude of Data		Equivalent Conversion
Not Less Than	Less Than	Effected
	0.1	Ew.d or Ew.dEe
0.1	1.0	F(w-n).d,n(' ')
1.0	10.0	F(w-n).(d-1),n(' ')
10.0	100.0	F(w-n).(d-2),n(' ')
.	.	.
.	.	.
.	.	.
10 ** (d-2)	10 ** (d-1)	F(w-n).1,n(' ')
10 ** (d-1)	10 ** d	F(w-n).0,n(' ')
10 ** d		Ew.d or Ew.dEe

The value of n is 4 for Gw.d format and (e+2) for Gw.dEe format. The n(' ') used in the above example indicates nth number of blanks. If the F form is chosen, then the scale factor is ignored. The following comparison between F formatting and G formatting is given by way of illustration:

Value	F F13.6 Conversion	G G13.6 Conversion
.01234567	0.012346	0.123457E-01
.12345678	0.123457	0.123457
1.23456789	1.234568	1.23457
12.34567890	12.345679	12.3457
123.45678900	123.456789	123.457
1234.56789000	1234.567890	1234.57
12345.67890000	12345.678900	12345.7
123456.78900000	123456.789000	123457.
1234567.89000000	**********	0.123457E+07

When an overflow condition occurs in a numeric field, the field is filled with asterisks (in the absence of any specification to the contrary by an overflow decoration), as shown above.

The "|" character is used to denote the boundaries of the output field.

NOTE

To use the G edit descriptor for output, floating-point firmware is required.

The I Edit Descriptor

The integer edit descriptor is used to display or interpret data values in an integer form.

The I edit descriptor has the following forms:

Iw OR Iw.m

w = an unsigned integer constant that defines the total width of the field and cannot exceed 255. The field processed is the w characters starting at the current position. After the field is processed, the current position is advanced by w characters.

m = an unsigned integer constant that defines the number of digits that must be present in the field on output.

On output, the external field consists of zero or more leading blanks (followed by a minus if the value of the internal data is negative, or an optional plus otherwise), followed by the magnitude of the internal value in the form of an unsigned integer constant without leading zeros. An integer constant always consists of at least one digit. The output from an Iw.m edit descriptor is the same as the above, except that the unsigned integer constant consists of at least m digits and, if necessary, has leading zeros. The value of m must not exceed the value of w. If m is zero and the internal data is zero, the output field consists only of blank characters, regardless of the sign control in effect; for example,

Format	Data Value	Result
I7	100	| 100|
I7.2	-1	| -01|
I7.6	100	| 000100|
I7.6	-1	|-000001|

The "|" character is used to denote the boundaries of the output field.

On input, an Iw.m edit descriptor is treated identically to an Iw edit descriptor. The edit descriptors Iw and Iw.m indicate that the field to be edited occupies w positions. In the input field, the character string must be in the form of an optionally signed integer constant, except for the interpretation of blanks. Leading blanks on input are not significant, and the interpretation of any other blanks is determined by blank control descriptors (BN and BZ); for example,

External Field	Format	Data Element Value
| 100|	I7	100
| -01|	I7	-1
| 1 |	I7	1
| 1 |	BZ,I7	1000
| 1 2 |	BZ,I7	10200
| 1 2 |	BN,I7	12

The "|" character is used to denote the boundaries of the output field.

The L Edit Descriptor

The logical edit descriptor is used to display or interpret data in logical form. The L edit descriptor has the form:

Lw

w = an unsigned integer constant that defines the width of the field and cannot exceed 255. The field processed is the w characters starting at the current position. After the field is processed, the current position is advanced by w characters.

On output, the L edit descriptor causes the associated data element to be evaluated in a logical context, and a single character is inserted right-justified in the output field. If the data value is null, the character is blank. If the data value is zero, the character is F; for all other cases, the character is T; for example,

Format	Data Value	Result
L2	-1	| T|
L2	15769	| T|
L2	0	| F|

The "|" character is used to denote the boundaries of the output field.

The input field consists of optional blanks, optionally followed by a decimal point, followed by an uppercase T for true (logical value -1) or an uppercase F for false (logical value 0). The T or F may be followed by additional characters in the field. The logical constants .TRUE. and .FALSE. are acceptable input forms; for example,

External Field	Format	Data Element Value
| T|	L7	-1
| F|	L7	0
| .TRUE.|	L7	-1
| .FALSE.|	L7	0
| TUGBOAT|	L7	-1
| FARLEY |	L7	0

The "|" character is used to denote the boundaries of the output field.

The M Edit Descriptor

The mask formatting edit descriptor edits either alphanumeric or numeric data according to an editing pattern or mask. Special characters within the mask indicate where digits in the data are to be displayed; other characters are duplicated in the output field as they appear in the mask.

The M edit descriptor has the form:

```
M<mask>

<mask> = a character string enclosed in a pair of
         apostrophes ('), a pair of quotation marks ("),
         or less-than and greater-than symbols (<>). The
         string supplied must not exceed 255 characters.
```

The M edit descriptor is not allowed for input.

Characters in a mask that have special functions are:

Z --Digit selector

9 --Digit selector

V --Decimal alignment character

. --Decimal alignment character

The field width w is determined by the number of characters, including spaces but excluding Vs, between the mask delimiters. The field processed is w characters starting at the current position. After the field is processed, the current position is advanced by w characters. Except for the decimal point alignment character, V, each character in the mask either defines a character position in the field or is directly inserted in the field.

The M edit descriptor causes numeric data elements to be rounded to the number of positions specified by the mask. String data elements are processed directly. Each digit or character of a data element is transferred to the result field in the next available character position that corresponds to a digit selector in the mask. If the digit selector is a 9, it causes the corresponding data digit to be transferred to the output field. The digit selector Z causes a nonzero, or embedded zero, digit

to be transferred to the field, but inserts blanks in place of leading or trailing zeros. Character positions must be allocated, by Z digit selectors, within the mask to provide for the inclusion of any minus signs or decoration character strings.

A decimal point in the mask can be used for decimal point alignment of the external field. The letter V can also be used for this purpose. If a V is present in the mask, the decimal point is located at the V, and the position occupied by the V is deleted. If no V is present, the decimal point is located at the rightmost occurrence of the decimal point character (usually "."). If neither a V nor a decimal point character is present, the decimal point is assumed to be to the right of the rightmost character of the entire mask.

Although leading and trailing text in a mask is always transferred to the result field, text embedded between digit selectors is transferred only if the corresponding digits to the right and left are transferred.

For example, a value that is intended to represent a date can be formatted with an M field descriptor as follows:

Format	Data Value	Result
M"99/99/99"	103179	|10/31/79|

The following is a comparison of the effects of using the 9 and Z as digit selectors. The minus sign in the preceding examples is the symbol that is automatically displayed for negative values in the absence of any specification to the contrary by a decoration. As shown in the preceding examples, a decimal point in the mask can be used for radix point alignment of the external field. Additional examples follow here:

Format	Data Values	Result
3M<Z99.99>	-27.40, 12, 0	-27.40 12.00 00.00
3M<ZZ9.99>	-27.40, 12, 0	-27.40 12.00 0.00
3M<ZZZ.99>	-27.40, 12, 0	-27.40 12.00 00

The "/\" marker is used to denote the boundaries of the output field.

In the example below, a comma specified as mask text is not displayed.

Format	Data Value	Result
M'Z,ZZ9.99'	32.009	| 32.01|

The "|"character is used to denote the boundaries of the output field.

Compare the different treatment of the embedded commas in the following examples:

```
Data Values:  298738472, 389487.987, 666, 0.35

Format One:  M<$ ZZZ,ZZZ,ZZ9 AND NO CENTS>

Format Two:  M<$ 999,999,999 AND NO CENTS>
```

Format One	Format Two
$ 298,738,472 AND NO CENTS	$ 298,738,472 AND NO CENTS
$ 389,488 AND NO CENTS	$ 000,389,488 AND NO CENTS
$ 666 AND NO CENTS	$ 000,000,666 AND NO CENTS
$ 0 AND NO CENTS	$ 000,000,000 AND NO CENTS

The M edit descriptor can be useful in producing visually effective reports, by formatting values into patterns that are meaningful in terms of the data they represent. For example, assume that four arrays contain the following data:

```
Amount    := 9758 21573 15532
Date      := 031777 091779 090579
District  := 'WEST','MIDWEST','SOUTH'
Telephone := 2135296800,2162296270,4047298400
```

The following format can then be used to output the data as a table whose entries are in familiar forms. Assuming the elements are presented to the formatter in the order: the first elements of each array, followed by the second elements of each array, and so on, using this format:

```
M<$ZZ,ZZ9>,M<   Z9/Z9/99>,3X,A8,M<   (999) 999-9999>
```

the result would be:

```
$ 9,758     3/17/77    WEST       (213) 529-6800
$21,573     9/17/79    MIDWEST    (216) 229-6270
$15,532     9/ 5/79    SOUTH      (404) 729-8400
```

MODIFIERS

Modifiers alter the normal effect of edit descriptors. Modifiers immediately precede the edit descriptor to which they apply. If modifiers immediately precede the left parenthesis of a group, the modifiers apply to each repeatable edit descriptor within the group. They are enclosed in brackets, and if more than one is present, they are separated by commas.

NOTE

Modifiers are effective only on output. If they are supplied for input, they have no effect.

Field-Blanking Modifiers (BN, BZ)

There are two modifiers for blanking fields:

BN = blank field if null.

BZ = blank field if equal to zero.

Although most edit descriptors cause a minimum number of characters to be output, a field-blanking modifier causes the entire field to be filled with spaces if the specified condition is met. The null value is the value addressed by the <nullptr> in the <variablelist> entry for the current data element.

Fill-Character Modifier (FL)

When an alphanumeric data element contains fewer characters than the field width specified by an Aw edit descriptor, when leading or trailing zero suppression is performed, or when embedded text in an M edit descriptor is not output because its neighboring digits are not output, a fill character is inserted in each appropriate character position in the output field. The fill character is normally a space, but the fill-character modifier can be used to specify any other character for this purpose.

The fill-character modifier has the form:

```
FL <char>

<char> = any single character, enclosed in quotation marks
         or apostrophes.
```

The following are examples of fill-character replacement:

Format	Data Value	Result
[FL'.']A10	'THEN'	|THEN.....|
[RJ,FL">"]A10	'HERE'	|>>>>>HERE|
[FL"*"]M<$ZZ,ZZ9.99>	127.39	|$***127.39|

The "|" character is used to denote the boundaries of the output field.

Overflow-Character Modifier (OC)

The overflow condition occurs if there are more characters to be placed into a field than there are positions provided by the edit descriptors. In the absence of any modifier or decoration to the contrary, if an overflow condition occurs in a numeric field, the field is filled with asterisks (*). This applies to the D, E, F, G, I, and M edit descriptors. The OC modifier can be used to substitute any other character for the asterisk as the overflow indicator character.

The OC modifier has the form:

```
OC <char>

<char> = any single character, enclosed in quotation
         marks or apostrophes.
```

For example, the modifier [OC '!'] causes the output field to be filled with exclamation marks, instead of asterisks, if an overflow occurs:

Format	Data Value	Results
[OC'!']I2	100	|!!|

The "|" character is used to denote the boundaries of the output field.

Justification Modifiers (LJ, RJ)

The A edit descriptor normally displays the data left justified in its field.

The justification modifiers are:

LJ - Left justify (normal)

RJ - Right justify (data is displayed right justified)

The RJ and LJ modifiers are used with the A edit descriptor only.

Symbol-Substitution Modifier (SS)

The symbol-substitution modifier permits the user to replace certain standard symbols used by the formatter with other symbols. It can be used with the M edit descriptor to free the special characters 9, V, ".", and Z for use as text characters in the mask. It can also be used with the D, E, F, and G edit descriptors to alter the standard characters they insert in the result field.

The symbol substitution modifier has the form:

SS <symprs>

<symprs> = one or more pairs of symbols enclosed in quotation marks or apostrophes. The first symbol in each pair is one of those in the following table; the second is the symbol that is to replace it temporarily.

The following formatting symbols can be altered by the SS modifier:

Symbol	Function
9	Digit selector (M format)
Z	Digit selector, zero suppression (M format)
V	Decimal alignment character (M format)
.	Decimal point (D, E, F, G, and M format)

The following examples show how the SS modifier can be used to permit decimal values to be displayed as clock times, to follow European conventions (where a comma is used as the decimal point and periods are used as digit group separators), or to alter the function of the digit selectors in the M edit descriptor. When using the symbol substitution with a mask format, to obtain the function of one special character which is being altered by the symbol substitution, use the new character of the pair. With all other formats, use the old character of the pair; for example:

Data Value	Format	Result
12.45	[SS".:"]F6.2	| 12:45|
12.45	[SS".:"]M<ZZZ:99>	| 12:45|
12345.67	[SS'.,']F10.2	| 12345,67|
103179	[SS<9X>]M<XX/XX/19XX>	|10/31/1979|

The "|" character is used to denote the boundaries of the output field.

The following table indicates which modifiers may be used with which edit descriptors (Y stands for yes, the combination is permitted).

		EDIT DESCRIPTORS						
		A	E,D	F	G	I	L	M
MODIFIERS	BZ,BN	Y	Y	Y	Y	Y	Y	Y
	LJ,RJ	Y						
	OC		Y	Y	Y	Y	Y	Y
	FL	Y	Y	Y	Y	Y		Y
	SS		Y	Y	Y			Y

DECORATIONS

A decoration specifies a character string that may be added to the result field, the conditions under which the string is to be added, the location at which the string is to be added, and whether it is to be added before normal formatting is done or after it is completed.

You can use multiple decorations, separated by commas, with the same edit descriptor. Decorations are enclosed in brackets (together with any modifiers) and immediately precede the edit descriptor to which they apply. If modifiers immediately precede the left parenthesis of a group, the modifiers apply to each repeatable edit descriptor within the group.

When a field is processed, the floating decorations appear in the same order, left to right. If an edit descriptor within a group already has some decorations, the decorations that are applied to the group function as if they were placed to the right of the decorations already present. A decoration has the form:

```
{ M }                                    { M }
{ N } ... { F } <string>     OR          { N }
{ P }     { P }                          { P } ... An <string>
{ Z }                                    { Z }
                                         { O }

Character 1 = Field condition specifier:  M - Minus
                                          N - Null
                                          O - Overflow
                                          P - Plus
                                          Z - Zero

Character 2 = String location specifier:  A - Absolute
                                          F - Floating
                                          P - Prior

          n = an unsigned nonzero integer constant that
              specifies the actual character position
              within the field at which the string is to
              begin.

   <string> = any character string enclosed in quotation
              marks or apostrophes.
```

NOTE

Only location type An can be used in combination with the O condition.

Conditions

The condition specifier states that the string is to be added to the field if its value is minus, zero, positive, or null, or if a field overflow has occurred. A null condition takes precedence over negative, positive, and zero conditions; the overflow test is done after those for the other conditions, and therefore precedence is not significant. Alphanumeric data elements are considered to be positive or null only.

A decoration may have more than one condition specifier. If multiple condition specifiers are entered, an "or" condition is understood. For example, "ZPA2'+' " specifies that the string is to be inserted in the field if the data value is equal to or greater than zero.

Locations

The location specifier indicates where the string is to be added to the field.

The A specifier states that the string is to begin in absolute position n within the field. The leftmost position of the field is position 1.

The F specifier states that, once the number of data characters in the field has been established, the string is to occupy the position or positions (for right-justified fields) immediately to the left of the leftmost data character. This is reversed for left-justified elements.

The P specifier states that, prior to normal formatting, the string is to be inserted in the rightmost (for right-justified fields) end of the field; data characters are shifted to the left an appropriate number of positions. This is reversed for left-justified fields.

Processing

Decoration processing is as follows:

1. The data element is determined to have a negative, positive, zero, or null value; a null condition takes precedence over the other attributes.

2. If a P location decoration is specified and its condition is satisfied, its string is inserted in the field.

3. Normal formatting is performed.

4. If A or F decorations are specified and their conditions met, they are applied.

5. If an attempt is made to transfer more characters to the field than can be accommodated (in step 2, 3, or 4), the overflow condition is set. If an overflow decoration has been specified, it is applied.

NOTE

Only location type An can be used with the O condition.

The following examples illustrate these considerations:

Format	Data Value	Result
[MF'<',MP'>',ZPP' ']F12.2	1000.00	1,000.00
[MF'<',MP'>',ZPP' ']F12.2	-1000.00	<1,000.00>
[MA1'CR',MPF'$']F12.2	1000.00	$1,000.00
[MA1'CR',MPF'$']F12.2	-100.00	CR $100.00
[OA1<**OVERFLOW**>]F12.2	1000000.00	1,000,000.00
[OA1<**OVERFLOW**>]F12.2	10000000.00	**OVERFLOW**

The "|" character is used to denote the boundaries of the output field.

NOTE

The following decorations are automatically applied to any numeric edit descriptor (D, E, F, G, I, or M) for which no decoration has been specified:

```
MF'-'
OA1'*** ... *'  (The number of asterisks is equal to the
                 number of characters in the field width.)
```

However, if any decoration with a condition code relating to the sign of the data is specified, the automatic "MF'-'" decoration no longer applies; if negative-value indication is desired, you must supply the appropriate decoration. If any decoration with a condition code relating to overflow is specified, the automatic "OA1'***...*'" decoration no longer applies.

As an example of how decorations apply to a group of edit descriptors, the following formats give the same results:

Format
[MF'-'](F10.2,[MZF'**']F10.2)
[MF'-']F10.2,[MZF'**',MF'-']F10.2

Using the format above:

Data Values	Results
0,0	0.00 **0.00
1,1	1.00 1.00
-1,-1	-1.00 **-1.00

The "/\" marker is used to denote the boundaries of the output field.

LIST-DIRECTED FORMATTING

List-directed formatting provides the data conversion capabilities of the formatter without requiring the specification of a format. The FORMATDATA procedure determines the details of the data conversion, based on the types of the data elements. This is particularly convenient for input because the list-directed formatting rules provide for free-format input of data values rather than requiring data to be supplied in fixed fields. There are fewer advantages to using list-directed formatting for output because the output data is not necessarily arranged in a convenient readable form.

The characters in one or more list-directed buffers constitute a sequence of data-list items and value separators. Each value is either a constant, a null value, or one of the following forms:

```
        r*c          r*

r is an unsigned, nonzero, integer constant.

r*c form is equivalent to r successive appearances of
the constant c.

r* form is equivalent to r successive null values.

Neither of these forms may contain embedded blanks,
except where permitted with the constant c.
```

List-Directed Input

All input forms that are acceptable to FORMATDATA when directed by a format are acceptable for list-directed input, with the following exceptions:

1. When the data element is a complex variable, the input form consists of a left parenthesis followed by an ordered pair of numeric input fields separated by a comma and followed by a right parenthesis.

2. When the data element is a logical variable, the input form must not include either slashes or commas among the optional characters for the L editing.

3. When the data element is a character variable, the input form consists of a string of characters enclosed in apostrophes. The blank, comma, and slash may appear in the string of characters.

4. A null value is specified by having no characters other than blanks between successive value separators, no characters preceding the first value separator in the first buffer, or the r* form. A null value has no effect on the value of the corresponding data element. The input list item retains its previous value. A single null value must represent an entire complex constant (not just part of it).

If a slash value separator is encountered during the processing of a buffer, data conversion is terminated. If there are additional elements in the data list, the effect is as if null values had been supplied for them.

On input, a value separator is one of the following:

1. A comma or slash optionally preceded or optionally followed by one or more contiguous blanks (except within a character constant).

2. One or more contiguous blanks between two constants or following the last constant (except embedded blanks surrounding the real or imaginary part of a complex constant).

3. The end of the buffer (except within a character constant).

List-Directed Output

Output forms that are produced by list-directed output are the same as that required for input with the following exceptions:

1. The end of a buffer may occur between the comma and the imaginary part of a complex constant only if the entire constant is as long as, or longer than, an entire buffer. The only embedded blanks permitted within a complex constant are between the comma and the end of a buffer, and one blank at the beginning of the next buffer.

2. Character values are displayed without apostrophes.

3. If two or more successive values in an output record produced have identical values, the FORMATDATA procedure produces a repeated constant of the form r*c instead of the sequence of identical values.

4. Slashes, as value separators, and null values are not produced by list-directed output.

For output, the value separator is a single blank. A value separator is not produced between or adjacent to character values.

APPENDIX A

PROCEDURE SYNTAX SUMMARY

The list on the following pages gives the abbreviated form of the calling syntax for the GUARDIAN operating system procedures and related Tandem software procedures callable by user programs. The full syntax for each of these procedures appears in the System Procedure Calls Reference Manual.

In the following procedure syntax overview, input parameters (those that pass data from the calling program to the called procedure) are followed by an "i" which indicates input. Output parameters (those that return data from the called procedure to the calling program) are followed by an "o" which indicates output. When a parameter can be both input and output, it is followed by "i, o".

```
CALL ABEND;

<status> := ABORTTRANSACTION;

CALL ACTIVATEPROCESS ( <process-id> );                    ! i

CALL ACTIVATERECEIVETRANSID ( <message-tag> );            ! i

CALL ADDDSTTRANSITION ( <low-gmt>                         ! i
                       ,<high-gmt>                        ! i
                       ,<offset>  );                      ! i
```

```
<status> := ALLOCATESEGMENT ( <segment-id>                    ! i
                            ,[ <segment-size> ]               ! i
                            ,[ <filename> ]                   ! i
                            ,[ <pin> ] );                     ! i

CALL ALTERPRIORITY ( <process-id>                             ! i
                   ,<priority> );                             ! i

CALL ARMTRAP ( <traphandlr-addr>                              ! i
             ,<trapstack-addr> );                             ! i

CALL AWAITIO ( <filenum>                                      ! i, o
             ,[ <buffer-addr> ]                               ! o
             ,[ <count-transferred> ]                         ! o
             ,[ <tag> ]                                       ! o
             ,[ <timelimit> ] );                              ! i

<status>  := BEGINTRANSACTION ( <trans-begin-tag> ) ;         !o

<contrl-chars> := BLINK^SCREEN ( @<screen-name>               ! i
                               ,SCREEN                        ! o
                               ,<buffer>                      ! i
                               ,<field-name>                  ! i
                               ,<blink>  );                   ! i

CALL CANCEL ( <filenum> );                                    ! i

CALL CANCELPROCESSTIMEOUT ( <tag> );                          ! i

CALL CANCELREQ ( <filenum>                                    ! i
               ,[ <tag> ]  );                                 ! i

CALL CANCELTIMEOUT ( <tag> );                                 ! i

CALL CHANGELIST ( <filenum>                                   ! i
                ,<function>                                   ! i
                ,<parameter> );                               ! i

CALL CHECKCLOSE ( <filenum>                                   ! i
                ,[ <tape-disposition> ]  );                   ! i
```

```
{ <status> := } CHECKMONITOR;
{ CALL       }


CALL CHECKOPEN ( <filename>                                  ! i
                ,<filenum>                                   ! i, o
                ,[ <flags> ]                                 ! i
                ,[ <sync or receive-depth> ]                 ! i
                ,[ <sequential-block-buffer> ]               ! i
                ,[ <buffer-length> ]                         ! i
                ,<backerror> );                              ! o


{ <status> := } CHECKPOINT
{ CALL       }

  ( [ <stack-base> [, [ <buffer-1> ] , [ <count-1> ] ]       ! i, i, i
                   [, [ <buffer-2> ] , [ <count-2> ] ]       ! i, i
                             .             .
                             .             .
                             .             .
                   [, [ <buffer-13>] , [ <count-13>] ] );    ! i, i


{ <status> := } CHECKPOINTMANY ( [ <stack-base> ]            ! i
{ CALL       }                  ,[ <descriptors> ] );        ! i


<error> := CHECK^SCREEN ( @<screen-name                      ! i
                         ,SCREEN                             ! o
                         ,<buffer>                           ! i
                         ,<check-procedure>                  ! o
                         ,<count>  );                        ! o


{ <status> := } CHECKSWITCH;
{ CALL       }


CALL CLOSE ( <filenum>                                       ! i
            ,[ <tape-disposition> ] );                       ! i


<jul-day-num> := COMPUTEJULIANDAYNO ( <year>                 ! i
                                     , <month>               ! i
                                     , <day>                 ! i
                                     ,[ <error-mask> ] );    ! o


<ret-timestamp> := COMPUTETIMESTAMP ( <date-n-time>          !  i
                                     ,[ <errormask> ]  );    ! o
```

```
CALL CONTIME ( <date-and-time>                                  ! o
             ,<t0>                                              ! i
             ,<t1>                                              ! i
             ,<t2> );                                           ! i

CALL CONTROL ( <filenum>                                        ! i
             ,<operation>                                       ! i
             ,<param>                                           ! i
             ,[ <tag> ]  );                                     ! i

CALL CONTROLBUF ( <filenum>                                     ! i
                ,<operation>                                    ! i
                ,<buffer>                                       ! i
                ,<count>                                        ! i
                ,[ <count-transferred> ]                        ! o
                ,[ <tag> ] );                                   ! i

CALL CONVERTPROCESSNAME ( <process-name> );                     ! i, o

CALL CONVERTPROCESSTIME ( <process-time>                        ! i
                        ,[ <hours> ]                            ! o
                        ,[ <minutes> ]                          ! o
                        ,[ <seconds> ]                          ! o
                        ,[ milliseconds> ]                      ! o
                        ,[ <microseconds> ]  );                 ! o

<ret-time> := CONVERTTIMESTAMP ( <julian-timestamp>             ! i
                               ,[ <direction> ]                 ! i
                               ,[ <node> ]                      ! i
                               ,[ <error> ] );                  ! o

CALL CPUTIMES ( [ <cpu> ]                                       ! i
              ,[ <sysid> ]                                      ! i
              ,[ <total-time> ]                                 ! o
              ,[ <cpu-process-busy> ]                           ! o
              ,[ <cpu-interrupt> ]                              ! o
              ,[ <cpu-idle> ]  );                               ! o
```

```
CALL CREATE ( <filename>                                        ! i, o
            ,[ <primary-extentsize> ]                           ! i
            ,[ <file-code> ]                                    ! i
            ,[ <secondary-extentsize> ]                         ! i
            ,[ <file-type> ]                                    ! i
            ,[ <recordlen> ]                                    ! i
            ,[ <data-blocklen> ]                                ! i
            ,[ <key-sequenced-params> ]                         ! i
            ,[ <alternate-key-params> ]                         ! i
            ,[ <partition-params> ]                             ! i
            ,[ <maximum-extents> ]                              ! i
            ,[ <unstructured-buffer-size ]                      ! i
            ,[ <open-defaults ]    );                           ! i

CALL CREATEPROCESSNAME ( <process-name> );                      ! o

CALL CREATEREMOTENAME ( <name>                                  ! o
                      ,<sysnum> );                              ! i

<accessor-id> := CREATORACCESSID;

{ <stack-env> := } CURRENTSPACE [ ( <ascii-space-id> ); ]       ! o
{ CALL          }

CALL DEALLOCATESEGMENT ( <segment-id>                           ! i
                       ,[ <flags> ] );                          ! o

CALL DEBUG;

CALL DEBUGPROCESS ( <process-id>                                ! i
                  ,<error>                                      ! o
                  ,[ <term> ]                                   ! i, o
                  ,[ <now> ]   );                               ! i, o

CALL DEFINELIST ( <filenum>                                     ! i
                ,<address-list>                                 ! i
                ,<address-size>                                 ! i
                ,<num-entries>                                  ! i
                ,<polling-count>                                ! i
                ,<polling-type> );                              ! i
```

```
<status> := DEFINEPOOL ( <pool-head>                          ! i, o
                        ,<pool>                               ! i
                        ,<pool-size> );                       ! i

CALL DELAY ( <time-period> );                                 ! i

CALL DEVICEINFO ( <filename>                                  ! i
                 ,<devtype>                                   ! o
                 ,<physical-recordlen> );                     ! o

CALL DEVICEINFO2 ( <filename>                                 ! i
                  ,<devtype>                                  ! o
                  ,<physical-recordlen>                       ! o
                  ,<discprocess-version> );                   ! o

<status> := EDITREAD ( <edit-controlblk>                      ! i
                      ,<buffer>                               ! o
                      ,<bufferlen>                            ! i
                      ,<sequence-num> );                      ! o

<status> := EDITREADINIT ( <edit-controlblk>                  ! i
                          ,<filenum>                          ! i
                          ,<bufferlen> );                     ! i

<status> := ENDTRANSACTION;

CALL ENFORMFINISH ( ctlblock );                               ! i

{ <count> := } ENFORMRECEIVE ( <ctlblock>                     ! i
{ CALL       }                ,<buffer> );                    ! i, o

CALL ENFORMSTART ( <ctlblock>                                 ! o
                  ,<compiled-physical-filename>               ! i
                  ,<buffer-length>                            ! i
                  ,<error-number>                             ! o
                  ,[ <restart-flag> ]                         ! i
                  ,[ <param-list> ]                           ! i
                  ,[ <assign-list> ]                          ! i
                  ,[ <qp-name> ]                              ! i
                  ,[ <cpu> ]                                  ! i
                  ,[ <priority> ]                             ! i
                  ,[ <timeout> ]                              ! i
                  ,[ <reserved-for-expansion> ]  );           ! i
```

```
<num-bytes> := EXPAND^SCREEN ( @<screen-name>              ! i
                             , SCREEN                      ! o
                             , <buffer>                    ! o
                             , <rewrite-form> );           ! i

<status> := FILEERROR ( <filenum> );                       ! i

CALL FILEINFO ( [ <filenum> ]                              !  i
              ,[ <error> ]                                 !  o
              ,[ <filename> ]                              !  i, o
              ,[ <ldevnum> ]                               !  o
              ,[ <devtype> ]                               !  o
              ,[ <extent-size> ]                           !  o
              ,[ <eof-location> ]                          !  o
              ,[ <next-record-pointer> ]                   !  o
              ,[ <last-modtime> ]                          !  o
              ,[ <filecode> ]                              !  o
              ,[ <secondary-extent-size> ]                 !  o
              ,[ <current-record-pointer> ]                !  o
              ,[ <open-flags> ]                            !  o
              ,[ <subdev> ]                                !  o
              ,[ <owner> ]                                 !  o
              ,[ <security> ]                              !  o
              ,[ <num-extents-allocated> ]                 !  o
              ,[ <max-file-size> ]                         !  o
              ,[ <partition-size> ]                        !  o
              ,[ <num-partitions> ]                        !  o
              ,[ <file-type> ]                             !  o
              ,[ <maximum-extents> ]                       !  o
              ,[ <unstructured-buffer-size> ]              !  o
              ,[ <open-flags2> ]                           !  o
              ,[ <sync-depth> ]                            !  o
              ,[ <next-open-fnum> ]  );                    !  o

CALL FILERECINFO ( [ <filenum> ]                           ! i
                 ,[ <current-keyspecifier> ]               ! o
                 ,[ <current-keyvalue> ]                   ! o
                 ,[ <current-keylen> ]                     ! o
                 ,[ <current-primary-keyvalue> ]           ! o
                 ,[ <current-primary-keylen> ]             ! o
                 ,[ <partition-in-error> ]                 ! o
                 ,[ <specifier-of-key-in-error> ]          ! o
                 ,[ <file-type> ]                          ! o
                 ,[ <logical-recordlen> ]                  ! o
                 ,[ <blocklen> ]                           ! o
                 ,[ <key-sequenced-parameters> ]           ! o
                 ,[ <alternate-key-parameters> ]           ! o
                 ,[ <partition-parameters> ]               ! o
                 ,[ <filename> ]  );                       ! i
```

```
CALL FIXSTRING ( <template>                                    ! i
               ,<template-len>                                 ! i
               ,<data>                                         ! i, o
               ,<data-len>                                     ! i, o
               ,[ <maximum-data-len> ]                         ! i
               ,[ <modification-status> ] );                   ! o

<length> := FL^SCREEN ( <field-name> );                        ! i

{ <length> := } FNAMECOLLAPSE ( <internal-name>                ! i
{ CALL       }                 ,<external-name> );             ! o

{ <status> := } FNAMECOMPARE ( <filename1>                     ! i
{ CALL       }                ,<filename2> );                  ! i

{ <length> := } FNAMEEXPAND ( <external-filename>              ! i
{ CALL       }               ,<internal-filename>              ! o
                             ,<default-names> );               ! i

{ <status> := } FORMATCONVERT ( <iformat>                      ! i
{ CALL       }                 ,<iformatlen>                   ! i
                               ,<eformat>                      ! o
                               ,<eformatlen>                   ! o
                               ,<scales>                       ! o
                               ,<scale-count>                  ! o, i
                               ,<conversion> );                ! i

{ <error> := } FORMATDATA ( <buffer>                           ! i, o
{ CALL      }               ,<bufferlen>                       ! i
                            ,<buffer-occurs>                   ! o
                            ,<length>                          ! o
                            ,<iformat>                         ! o
                            ,<variable-list>                   ! o
                            ,<variable-list-len>               ! o
                            ,<flags> );                        ! i

CALL GETCPCBINFO ( <request-id>                                ! i
                 ,<cpcb-info>                                  ! o
                 ,<out-length>                                 ! i
                 ,<out-length>                                 ! o
                 ,<error>  );                                  ! o

CALL GETCRTPID ( <cpu,pin>                                     ! i
               ,<process-id> );                                ! o
```

```
<status> := GETDEVNAME ( <ldevnum>                          ! i, o
                        ,<devname>                          ! o
                        ,[ <sysnum> ]  );                   ! i

<address> := GETPOOL ( <ppd-head>                           ! i, o
                      ,<block-size>  );                     ! i

CALL GETPPDENTRY ( <index>                                  ! i
                  ,<sysnum>                                 ! i
                  ,<ppd>                                    ! o

CALL GETREMOTECRTPID ( <pid>                                ! i
                      ,<process-id>                         ! o
                      ,<sysnum> );                          ! i

CALL GETSYNCINFO ( <filenum>                                ! i
                  ,<sync-block>                             ! o
                  ,[ <sync-block-size> ] );                 ! o

{ <ldev> := } GETSYSTEMNAME ( <sysnum>                      ! i, o
{ CALL     }                ,<sysname> );                   ! o

<status>  := GETTMPNAME ( <devname> );                      ! o

<status> := GETTRANSID ( <transid> );                       ! i

CALL HALTPOLL ( <filenum> );                                ! i

CALL HEAPSORT ( <array>                                     ! i, o
               ,<num-elements>                              ! i
               ,<size-of-element>                           ! i
               ,<compare-proc> );                           ! i

{ <status> := } INITIALIZER ( [ <rucb> ]                    ! i
{ CALL       }                ,[ <passthru> ]               ! o
                              ,[ <startupproc> ]            ! i
                              ,[ <paramsproc> ]             ! i
                              ,[ <assignproc> ]             ! i
                              ,[ <flags> ]  );              ! i
```

```
CALL INTERPRETJULIANDAYNO ( <julian-day-num>                    ! i
                          ,<year>                               ! o
                          ,<month>                              ! o
                          ,<day>  );                            ! o

<ret-date-time> := INTERPRETTIMESTAMP ( <julian-timestamp>      ! i
                                      , <date-n-time>  );       ! o

<retval> := JULIANTIMESTAMP ( [ <type> ]                        ! i
                            ,[ <tuid> ]  );                     ! o

CALL KEYPOSITION ( <filenum>                                    ! i
                 ,<key-value>                                   ! i
                 ,[ <key-specifier> ]                           ! i
                 ,[ <length-word> ]                             ! i
                 ,[ <positioning-mode> ] );                     ! i

<last-addr> := LASTADDR;

CALL LASTRECEIVE ( [ <process-id> ]                             ! o
                 ,[ <message-tag> ] );                          ! o

{ <ldev> := } LOCATESYSTEM ( <sysnum>                           ! i, o
{ CALL     }               ,[ <sysname>] );                     ! i

CALL LOCKFILE ( <filenum>                                       ! i
              ,[ <tag> ]  );                                    ! o

CALL LOCKREC ( <filenum>                                        ! i
             ,[ <tag> ] );                                      ! i

CALL LOOKUPPROCESSNAME ( <ppd> );                               ! i, o

CALL MOM ( <process-id> );                                      ! o

CALL MONITORCPUS ( <cpu-mask> );                                ! i

CALL MONITORNET ( <enable> );                                   ! i

CALL MONITORNEW ( <enable> );                                   ! i
```

```
<cpu,pin> := MYPID;

<process-time> := MYPROCESSTIME;

<sysnum> := MYSYSTEMNUMBER;

CALL MYTERM ( <filename> );                                    ! o

CALL NEWPROCESS ( <filenames>                                  ! i
                 ,[ <priority> ]                               ! i
                 ,[ <memory-pages> ]                           ! i
                 ,[ <processor> ]                              ! i
                 ,[ <process-id> ]                             ! o
                 ,[ <error> ]                                  ! o
                 ,[ <name> ]                                   ! i
                 ,[ <hometerm> ]                               ! i
                 ,[ <inspect-flag> ]   );                      ! i

CALL NEWPROCESSNOWAIT ( <filenames>                            ! i
                       ,[ <priority> ]                         ! i
                       ,[ <memory-pages> ]                     ! i
                       ,[ <processor> ]                        ! i
                       ,[ <process-id> ]                       ! unused
                       ,[ <error> ]                            ! o
                       ,[ <name> ]                             ! i
                       ,[ <hometerm> ]                         ! i
                       ,[ <inspect-flags> ]  );                ! i

<error> := NEXTFILENAME ( <filename> );                        ! i, o

{ <next-addr> := } NUMIN ( <ascii-num>                         ! i
{ CALL           }        ,<signed-result>                     ! o
                          ,<base>                              ! i
                          ,<status> );                         ! o

CALL NUMOUT ( <ascii-result>                                   ! o
             ,<unsigned-integer>                               ! i
             ,<base>                                           ! i
             ,<width> );                                       ! i
```

```
CALL OPEN ( <filename>                                          ! i
          ,<filenum>                                            ! o
          ,[ <flags> ]                                          ! i
          ,[ <sync-or-receive-depth> ]                          ! i
          ,[ <primary-filenum> ]                                ! o
          ,[ <primary-process-id> ]                             ! o
          ,[ <seq-block-buffer> ]                               ! i unused
          ,[ <buffer-length ]  );                               ! i


CALL POSITION ( <filenum>                                       ! i
              ,<record-specifier> );                            ! i


<num-chars> := POSITION^SCREEN ( @<screen-name>                 ! i
                               , SCREEN                         ! o
                               , <buffer>                       ! o
                               , <field-name> );                ! i


<error-code> := PRINTCOMPLETE ( <filenum-to-supervisor>           ! i
                              ,<print-control-buffer>  );         ! o


<error-code> := PRINTINFO ( <job-buffer>                        ! i
                          ,[ <copies-remaining> ]               ! o
                          ,[ <current-page> ]                   ! o
                          ,[ <current-line> ]                   ! o
                          ,[ <lines-printed> ] );               ! o


<error-code> := PRINTINIT ( <filenum-to-supervisor>             ! i
                          ,<print-control-buffer> );            ! i, o


<error-code> := PRINTREAD ( <job-buffer>                        ! i, o
                          ,<data-line>                          ! o
                          ,<read-count>                         ! i
                          ,[ <count-read> ]                     ! o
                          ,[ <pagenum> ]   ) ;                  ! i
```

```
<error-code> := PRINTREADCOMMAND ( <print-control-buffer> ! i
                                 ,[ <controlnum> ]        ! o
                                 ,[ <device> ]            ! o
                                 ,[ <devflags> ]          ! o
                                 ,[ <devparam> ]          ! o
                                 ,[ <devwidth> ]          ! o
                                 ,[ <skipnum> ]           ! o
                                 ,[ <data-file> ]         ! o
                                 ,[ <jobnum> ]            ! o
                                 ,[ <location> ]          ! o
                                 ,[ <form-name> ]         ! o
                                 ,[ <report-name> ]       ! o
                                 ,[ <pagesize> ]   );     ! o

<error-code> := PRINTSTART ( <job-buffer>                 ! o
                           ,<print-control-buffer>        ! i
                           ,<data-filenum>  );            ! i

<error-code> := PRINTSTATUS ( <filenum-to-supervisor>     ! i
                            ,<print-control-buffer>       ! i
                            ,<msg-type>                   ! i
                            ,<device>                     ! i
                            ,[ <error> ]                  ! i
                            ,[ <num-copies> ]             ! i
                            ,[ <page> ]                   ! i
                            ,[ <line> ]                   ! i
                            ,[ <lines-printed> ] );       ! i

{ <old-priority> := } PRIORITY ( [ <new-priority> ]       ! i
{ CALL             }             ,[ <init-priority> ] );  ! o

<accessor-id> := PROCESSACCESSID;

<old-security> := PROCESSFILESECURITY ( <security> );     ! i
```

```
{ <error> := } PROCESSINFO ( <cpu,pin>                          ! i
{ CALL       }               ,[ <process-id> ]                  ! i, o
                             ,[ <creator-accessor-id> ]         ! i, o
                             ,[ <process-accessor-id> ]         ! i, o
                             ,[ <priority> ]                    ! i, o
                             ,[ <program-filename> ]            ! i, o
                             ,[ <home-terminal> ]               ! i, o
                             ,[ <sysnum> ]                      ! i
                             ,[ <search-mode> ]                 ! i
                             ,[ <priv-only> ]                   ! i
                             ,[ <processtime> ]                 ! o
                             ,[ <waitstate> ]                   ! o
                             ,[ <process-state> ]               ! o
                             ,[ <library-filename> ]            ! o
                             ,[ <swap-filename> ]  );           ! o

<processor-status> := PROCESSORSTATUS;

<type> := PROCESSORTYPE ( [ <cpu> ]                             ! i
                         ,[ <sysid> ]  );                       ! i

<process-time> := PROCESSTIME ( [ <cpu,pin> ]                   ! i
                               ,[ <sysid> ] );                  ! i

CALL PROGRAMFILENAME ( <program-file> );                        ! o

CALL PURGE ( <filename> );                                      ! i

CALL PUTPOOL ( <pool-head>                                      ! i, o
              ,<pool-block> );                                  ! i

CALL READ ( <filenum>                                           ! i
           ,<buffer>                                            ! o
           ,<read-count>                                        ! i
           ,[ <count-read> ]                                    ! o
           ,[ <tag> ]  );                                       ! i

CALL READLOCK ( <filenum>                                       ! i
               ,<buffer>                                        ! o
               ,<read-count>                                    ! i
               ,[ <count-read> ]                                ! i
               ,[ <tag> ]  );                                   ! i
```

```
<num-chars> :=  READ^SCREEN ( @<screen-name>                    ! i
                            , <buffer> );                       ! o

CALL READUPDATE ( <filenum>                                     ! i
                 ,<buffer>                                      ! o
                 ,<read-count>                                  ! i
                 ,[ <count-read> ]                              ! o
                 ,[ <tag> ]  );                                 ! i

CALL READUPDATELOCK ( <filenum>                                 ! i
                     ,<buffer>                                  ! o
                     ,<read-count>                              ! i
                     ,[ <count-read> ]                          ! o
                     ,[ <tag> ]  );                             ! i

CALL RECEIVEINFO ( [ <process-id> ]                             ! o
                  ,[ <message-tag> ]                            ! o
                  ,[ <sync-id> ]                                ! o
                  ,[ <filenum> ]                                ! o
                  ,[ <read-count> ] );                          ! o

{ <error> := } REFRESH  ( [ <volname> ]                         ! i
{  CALL      }            ,[ <all> ]  );                        ! i

<status> := REMOTEPROCESSORSTATUS ( <sysnum> );                 ! i

<tos-version> := REMOTETOSVERSION ( [ <sysid> ] );              ! i

CALL RENAME ( <filenum>                                         ! i
            ,<new-name> );                                      ! i

CALL REPLY ( [ <buffer> ]                                       ! i
            ,[ <write-count> ]                                  ! i
            ,[ <count-written> ]                                ! o
            ,[ <message-tag> ]                                  ! i
            ,[ <error-return> ] );                              ! i

CALL REPOSITION ( <filenum>                                     ! i
                 ,<positioning-block> );                        ! i

CALL RESERVELCBS ( <no-receive-lcbs>                            ! i
                  ,<no-send-lcbs>  );                           ! i
```

```
CALL RESETSYNC ( <filenum> );                                    ! i

<status> := RESUMETRANSACTION ( <trans-begin-tag> );             ! i

CALL SAVEPOSITION ( <filenum>                                    ! i
                  ,<positioning-block>                           ! o
                  ,[ <positioning-blksize> ] );                  ! o

CALL SETLOOPTIMER ( <new-time-limit>                             ! i
                  ,[ <old-time-limit> ] );                       ! o

CALL SETMODE ( <filenum>                                         ! i
             ,<function>                                         ! i
             ,[ <param1> ]                                       ! i
             ,[ <param2> ]                                       ! i
             ,[ <last-params> ]  );                              ! o

CALL SETMODENOWAIT ( <filenum>                                   ! i
                   ,<function>                                   ! i
                   ,[ <param1> ]                                 ! i
                   ,[ <param2> ]                                 ! i
                   ,[ <last-params> ]                            ! o
                   ,[ <tag>]  );                                 ! i

CALL SETMYTERM ( <terminal-name> );                              ! i

CALL SETPARAM ( <filenum>                                        ! i
              ,<function>                                        ! i
              ,[ <param-array> ]                                 ! i
              ,[ <param-count> ]                                 ! i
              ,[ <last-param-array> ]                            ! o
              ,[ <last-param-count> ] );                         ! o

{ <last-stop-mode> := } SETSTOP ( <stop-mode> );                 ! i
{ CALL                 }

CALL SETSYNCINFO ( <filenum>                                     ! i
                 ,<sync-block> );                                ! o

CALL SETSYSTEMCLOCK ( <julian-gmt>                               ! i
                    , <mode>                                     ! i
                    ,[ <tuid>]  );                               ! i
```

```
CALL SHIFTSTRING ( <string>                                    ! i, o
                  ,<count>                                     ! i
                  ,<casebit> );                                ! i


CALL SIGNALPROCESSTIMEOUT ( <timeout-value>                    ! i
                           ,[ <param1> ]                       ! i
                           ,[ <param2> ]                       ! i
                           ,[ <tag> ]  );                      ! o


CALL SIGNALTIMEOUT ( <timeout-value>                           ! i
                    ,[ <param1> ]                              ! i
                    ,[ <param2> ]                              ! i
                    ,[ <tag> ] );                              ! o


{ <status> := } SORTERROR ( <ctlblock>                         ! i
{ CALL       }             ,<buffer>  );                       ! i


{ <status> := } SORTERRORDETAIL ( <ctlblock> );                ! i
{ CALL       }


{ <status> := } SORTMERGEFINISH ( <ctlblock>                   ! i, o
{ CALL       }                   ,[ <abort> ]                  ! i
                                 ,[ <spare1> ]      ! error if used.
                                 ,[ <spare2> ] );   ! error if used.


{ <status> := } SORTMERGERECEIVE ( <ctlblock>                  ! i
{ CALL       }                    ,<buffer>                    ! i
                                  ,<length>                    ! o
                                  ,[ <spare1> ]     ! error if used.
                                  ,[ <spare2> ] );  ! error if used.


{ <status> := } SORTMERGESEND ( <ctlblock>                     ! i
{ CALL       }                 ,<buffer>                       ! i
                               ,<length>                       ! i
                               ,[ <stream id> ]                ! i
                               ,[ <spare1> ]        ! error if used.
                               ,[ <spare2> ] );     ! error if used.
```

```
{ <status> := } SORTMERGESTART ( <ctlblock>                      ! i
{ CALL        }                 ,<key-block>                     ! i
                                ,[ <num-merge-files> ]           ! i
                                ,[ <num-sort-files> ]            ! i
                                ,[ <in-filename> ]               ! i
                                ,[ <in-file-exclusion-mode> ]    ! i
                                ,[ <in-file-count> ]             ! i
                                ,[ <in-file-length> ]            ! i
                                ,[ <format> ]                    ! i
                                ,[ <out-filename> ]              ! i
                                ,[ <out-file-exclusion-mode> ]   ! i
                                ,[ <out-file-type> ]             ! i
                                ,[ <flags> ]                     ! i
                                ,[ <errnum> ]                    ! o
                                ,[ <errproc> ]                   ! i
                                ,[ <scratch-filename> ]          ! i
                                ,[ <scratch-block> ]             ! i
                                ,[ <process-start> ]             ! i
                                ,[ <max-record-length> ]         ! i
                                ,[ <collate-sequence-table> ]    ! i
                                ,[ <sparel> ]                    ! i
                                ,[ <spare2> ]                    ! i
                                ,[ <spare3> ]                    ! i
                                ,[ <spare4> ]       ! error if used
                                ,[ <spare5> ]  );   ! error if used

{ <status> := }  SORTMERGESTATISTICS ( <ctlblock>          ! i
{ CALL        }                      ,<length>             ! i, o
                                     ,<statistics>         ! o
                                     ,[ <sparel> ]   ! do not use
                                     ,[ <spare2 ] ); ! do not use


<error-code> := SPOOLCONTROL ( <level-3-buff>              ! i, o
                             ,<operation>                  ! i
                             ,<param>                      ! i
                             ,[ <bytes-written-to-buff> ] );! o


<error-code> := SPOOLCONTROLBUF ( <level-3-buff>           ! i, o
                                ,<operation>               ! i
                                ,<buffer>                  ! i
                                ,<count>                   ! i
                               ,[ <bytes-written-to-buff> ] ); ! o


<error-code> := SPOOLEND ( <level-3-buff>                  ! i
                         ,[ <flags> ] );                   ! i
```

```
<error-code> := SPOOLERCOMMAND ( <filenum-to-supervisor>  ! i
                               ,<command-code>            ! i
                               ,[ <command-param> ]       ! i
                               ,<subcommand-code>         ! i
                               ,[ <subcommand-param> ] ); ! i

<error-code> := SPOOLEREQUEST ( <supervisor-filenum>      ! i
                              ,<job-num>                  ! i
                              ,<print-control-buffer> );  ! o

<error-code> := SPOOLERSTATUS ( <supervisor-filenum>      ! i
                               ,<command-code>            ! i
                               ,<scan-type>               ! i
                               ,<status-buffer> );        ! i, o

<error-code> := SPOOLJOBNUM ( <filenum-to-collector>      ! i
                            ,<job-num> );                 ! o

<error-code> := SPOOLSETMODE ( <level-3-buff>             ! i, o
                             ,<function>                  ! i
                             ,[ <param1> ]                ! i
                             ,[ <param2> ]                ! i
                             ,[ <bytes-written-to-buff> ] ); ! o

<error-code> := SPOOLSTART ( <filenum-to-collector>       ! i
                           ,[ <level-3-buff> ]            ! o
                           ,[ <location> ]                ! i
                           ,[ <form-name> ]               ! i
                           ,[ <report-name> ]             ! i
                           ,[ <num-of-copies> ]           ! i
                           ,[ <page-size> ]               ! i
                           ,[ <flags> ]                   ! i
                           ,[ <owner> ] );                ! i

<error-code> := SPOOLWRITE ( <level-3-buff>               ! i, o
                           ,<print-line>                  ! i
                           ,<write-count>                 ! i
                           ,[ <bytes-written-to-buff> ] ); ! o

CALL STEPMOM ( <process-id> );                            ! i

CALL STOP [ ( <process-id>                                ! i
            ,[ <stop-backup> ] );                         ! i
```

```
CALL SUSPENDPROCESS ( <process-id> );                          ! i

<label> := SYSTEMENTRYPOINTLABEL ( <name>                      ! i
                                  ,<len> );                    ! i

CALL TIME ( <date-and-time> );                                 ! o

CALL TIMESTAMP ( <interval-clock> );                           ! o

<version> := TOSVERSION;

CALL UNLOCKFILE ( <filenum>                                    ! i
                 ,[ <tag> ]   );                               ! i

CALL  UNLOCKREC ( <filenum>                                    ! i
                ,[ <tag> ]  );                                 ! i

CALL USERIDTOUSERNAME ( <id-name> );                           ! i, o

CALL USERNAMETOUSERID ( <name-id> );                           ! i, o

<old-segment-id> := USESEGMENT ( <segment-id> );               ! i

CALL VERIFYUSER ( <user-name-or-id>                            ! i
                 ,[ <logon> ]                                  ! i
                 ,[ <default> , <default-len> ]  );            ! o, i

CALL WRITE ( <filenum>                                         ! i
            ,<buffer>                                          ! i
            ,<write-count>                                     ! i
            ,[ <count-written> ]                               ! o
            ,[ <tag> ]  );                                     ! i

CALL WRITEREAD ( <filenum>                                     ! i
                ,<buffer>                                      ! i,o
                ,<write-count>                                 ! i
                ,<read-count>                                  ! i
                ,[ <count-read> ]                              ! o
                ,[ <tag> ]  );                                 ! i
```

```
CALL WRITEUPDATE ( <filenum>                                    ! i
                  ,<buffer>                                     ! i
                  ,<write-count>                                ! i
                  ,[ <count-written> ]                          ! o
                  ,[ <tag> ]  );                                ! i

CALL WRITEUPDATEUNLOCK ( <filenum>                              ! i
                        ,<buffer>                               ! i
                        ,<write-count>                          ! i
                        ,[ <count-written> ]                    ! o
                        ,[ <tag> ]  );                          ! i
```

SIO Procedures Syntax

```
<state> := CHECK^BREAK ( { <common-FCB> } );                    ! i
                         { <file-FCB>   }                       ! i

<retval> := CHECK^FILE ( { <common-FCB> }                       ! i
                         { <file-FCB>   }                       ! i

  ,<operation>  );                                              ! i

{ <error> := } CLOSE^FILE ( { <common-FCB> }                    ! i
{ CALL      }               { <file-FCB>   }                    ! i

  [, <tape-disposition> ]  );                                   ! i

{ <error> := } GIVE^BREAK ( { <common-FCB> } );                 ! i
{ CALL      }               { <file-FCB>   }                    ! i

{ <no-retry> := } NO^ERROR ( <state>                            ! i
{ CALL         }             ,<file-fcb>                        ! i
                             ,<good-error-list>                 ! i
                             ,<retryable>  );                   ! i

{ <error> := } OPEN^FILE ( <common-fcb>                         ! i
{ CALL      }             ,<file-fcb>                           ! i
                          ,[ <block-buffer> ]                   ! i
                          ,[ <block-bufferlen> ]                ! i
                          ,[ <flags> ]                          ! i
                          ,[ <flags-mask> ]                     ! i
                          ,[ <max-recordlen> ]                  ! i
                          ,[ <prompt-char> ]                    ! i
                          ,[ <error-file-fcb> ]  );             ! i
```

SIO Syntax Summary

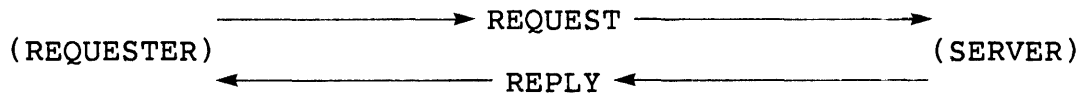
```
{ <error> := } READ^FILE ( <file-fcb>                    ! i
{ CALL       }             ,<buffer>                     ! o
                           ,[ <count-read> ]             ! o
                           ,[ <prompt-count> ]           ! i
                           ,[ <max-read-count> ]         ! i
                           ,[ <nowait> ]  );             ! i

{ <error> := } TAKE^BREAK ( <file-fcb> );                ! i
{ CALL       }

{ <error> := } WAIT^FILE ( <file-fcb>                    ! i
                           ,[ <count-read> ]             ! o
                           ,[ <time-limit> ]  );         ! i

{ <error> := } WRITE^FILE ( <file-fcb>                    ! i
{ CALL       }              ,<buffer>                     ! i
                            ,<write-count>                ! i
                            ,[ <reply-error-code> ]       ! i
                            ,[ <forms-control-code> ]     ! i
                            ,[ <nowait> ]   );            ! i
```

APPENDIX B

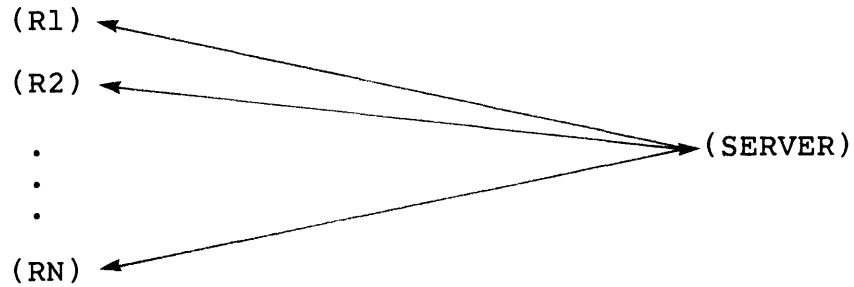
FAULT-TOLERANT PROGRAMMING EXAMPLE

This section presents an example of programming technique involving the use of NonStop process pairs. The example program is a NonStop server process.

A "server" process is an application process that accepts a request from a "requester" process, fulfills the request, then returns a reply (usually consisting of a data message or an error indication) to the requester:



Typically, a server process can accept requests from multiple requester processes:



Server processes are used in instances where

- it is desired to modularize the application by function.
- several application programs need to execute complex, but similar functions. In this case, a server process can perform these functions on behalf of the requester processes. This eliminates the need for each program to contain large amounts of similar coding.

- it is desired that each terminal in an application be controlled by a separate process, and several processes must access the same set of disc files, but it is undesirable for each process to have the disc files open. In this case, a server process performs all the disc operations, and therefore is the only process with the disc files open. The terminal requester processes perform disc operations by sending requests to the server process using interprocess communication methods. This method of disc file handling can also be used to reduce or eliminate the need for file locking.

- a custom interface to a nonstandard I/O device is needed. In this case, the server process translates normal file system WRITE or WRITEREAD requests into the specific file system requests needed to control the device. An example of this is a server process that interfaces, via ENVOY protocols, to a data communications line; requests are made to the server as though communicating with a conversational mode terminal (through WRITEREAD), and the server translates the request into the WRITEs, READs, and CONTROLs needed to control the line.

The example presented here is a server process that accesses a data base on behalf of several requester processes. As such, the example program is a culmination of the programming techniques discussed in Section 4, "Communicating With Other Processes", and Section 12, "Writing Fault-Tolerant Programs".

EXAMPLE PROGRAM

The example program (Figure B-1) is called "serveobj", and its source program is called "servesrc". It executes as a process pair in two processor modules. One process of the pair is the primary process; it performs the requested operations. The other process of the pair is the backup process; it monitors the operational state of the primary. If the primary server process becomes inoperable, the backup process takes over and performs the server function.

When the "servesrc" program is initially run (by a command interpreter RUN command), it is given the process name "$SERVE" (requester processes access the server by this name). The first process created assumes the role of the primary process. The primary process, after successfully opening its files, creates the backup process, then opens the backup's files. The backup process, as soon as it determines that it is the backup, begins monitoring the primary process.

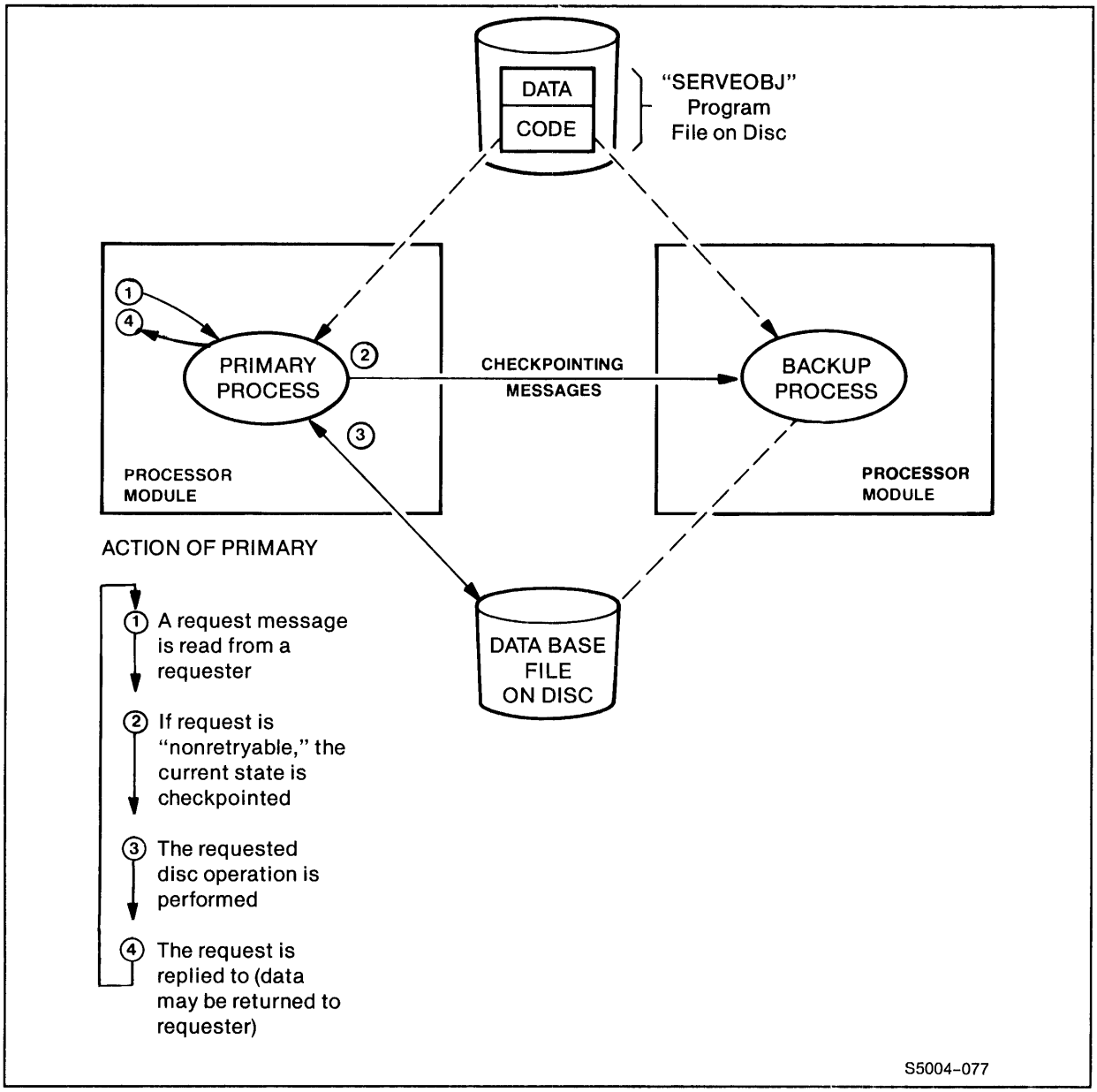


Figure B-1. "Serveobj" Program

The function of the primary process is to read a request message, perform the action indicated by the request, then return the outcome of the request--an error code and/or data--in a reply message to the requester. The requests that the server processes are

- insert: Insert the supplied record into the data base.
- delete: Delete the supplied record from the data base.

- query: Return the record from the data base as indicated by the supplied key value.
- next: Return the next record from the data base as indicated by the supplied key value.

Example Program Structure

The example program consists of the following procedures:

- "serve"

  This is the main procedure. It is where the primary/backup determination is made. If the process is the primary, the startup message is read, the data base file is opened, the backup is created, and the main processing loop is called. If the process is the backup, the CHECKMONITOR procedure is called.

- "execute"

  This is the execution loop of the server process. It waits on the $RECEIVE file for incoming messages. If a user request message is received, the "process^user^request" procedure is called. If a system message is received, the "analyze^system^message" procedure is called.

- "process^user^request"

  This procedure interprets the user request, checks for duplicate requests, checkpoints in some cases, calls the appropriate "primitive" to process the user request, and saves the outcome of the operation for the requester. The user request primitives are

  --"query^request": get the record from the data base as indicated by the supplied key value.

  --"insert^request": insert the supplied record into the data base.

  --"delete^request": delete the record from the data base as indicated by the supplied key value.

  --"next^record^request": get the next record from the data base as indicated by the supplied key value.

- "analyze^system^message"

  This procedure interprets system messages and takes appropriate action. This involves creating a backup process in some cases, and adding and deleting requesters from the "requester process directory".

  The "requester process directory" is a table of all processes (and their backups) currently accessing the server. This is maintained so that the server can ensure that the proper reply is made to a requester in the event of a requester or server failure (see "Request Integrity"). There are four "primitives" used to perform directory operations:

  --"lookuppid": look up a requester in the directory (called when a user request message is received).

  --"addpid": add a requester to the directory (called when an OPEN system message is received).

  --"delpid": delete a requester from the directory (called when a CLOSE system message is received).

  --"delallpids": delete all requesters from the directory that are associated with a given CPU (called if a CPU failure occurs).

- "analyze^checkpoint^status"

  This procedure is called when a nonzero return is made from the CHECKPOINT procedure. This usually occurs only when the backup takes over. This procedure analyzes the reason for the takeover and takes appropriate action.

- "create^backup"

  This procedure performs the backup process creation function. It is called at the start of primary execution, called when the primary detects a failure of the backup and the backup processor module is operable, and called on a takeover by the backup when the primary process failed because of an ABEND condition. Following a successful creation of the backup process, the files are opened for the backup, and the current state of the primary process is checkpointed.

- "open^primarys^files" and "open^backups^files"

  These two procedures perform the file open functions for the primary process and the backup process, respectively.

- "read^start^up^message"

  This procedure is called at the beginning of the primary process's execution to open the $RECEIVE file, read the startup message, and save the file name of the data base disc file.

Request Integrity

For the purpose of preventing erroneous results being returned to a requester if a failure of a requester or the server occurs, requests are classified by the server as being either retryable or nonretryable. The "retryable" requests are those which do not alter the data base, and therefore can be reexecuted indefinitely with the same results. The retryable requests are "query" and "next". The "nonretryable" requests, conversely, alter the data base and would return different results if reexecuted (for example, the first insert of a given record is successful, but the second insert of the same record results in a "record already exists" error). The nonretryable requests are "insert" and "delete".

Each request message contains a sync ID. The sync ID is used by the server process to detect duplicate requests for nonretryable operations (such as "insert" or "delete") from a given requester process (duplicate requests are caused by a failure of a requester process). The value of the sync ID is incremented by the requester with each request. When a new request is received, and the request is nonretryable, the server saves the value of the sync ID (these are saved for each given requester). If the sync ID in a request does not match the saved sync ID for the requester, then the request is a new request. In this case, the requested operation is performed, the results are returned to the requester, and the error code which was returned to the requester (to indicate the outcome of the operation) is saved by the server process for the requester. (Note that, because the server only saves the sync ID's associated with nonretryable requests, the sync ID associated with a retryable request will always indicate a new request. Therefore, duplicate retryable requests will be reexecuted by the server). If the sync ID in a request message matches the saved sync ID, then the request is a duplicate request for a nonretryable operation. The server does not reexecute the operation. Rather, it returns the completion status that it saved for that requester.

Checkpoints

There are three types of checkpoints in the example program:

- initial checkpoint
- request checkpoint
- process requester directory checkpoint

INITIAL CHECKPOINT: The initial checkpoint is made to the backup process, in the "create^backup" procedure", following the successful creation of the backup and the opening of its files. The checkpoint includes the entire data area from 'G'[5] through the top-of-stack location, and includes the "sync block" for the data base file (the variable "backup^cpu" is not checkpointed). At this point, the backup process's data area is an exact copy of the primary process's data area.

REQUEST CHECKPOINT (Figure B-2): Each time through its main processing loop in the "process^user^request" procedure, if the current request is nonretryable, the primary checkpoints the outcome of the preceding nonretryable request and the state of the current request to the backup. This is accomplished via a call to the CHECKPOINT procedure. The purpose of the checkpoint is to keep the backup informed as to the current state of the primary and to define a restart point for the backup in the event that the primary fails.

In the example program, the goal was to keep the number of checkpoints and the amount of data checkpointed to a minimum. It is important to note that the checkpoint is made only if the current request is nonretryable, and that this one checkpoint per processing loop is ample for all failure recovery. The data checkpointed is:

- the data stack. This is kept small by the use of global data buffers.
- the data base file sync block
- the sync ID, by requester, of the current request
- the data record
- the error return value, by requester, of the preceding nonretryable request

Any time a failure of the server primary occurs, the backup takes over from the latest checkpoint (which is for the latest nonretryable operation) and reexecutes the latest nonretryable operation. (Note that this generates the result value for the latest nonretryable operation. The backup now has the correct values for any nonretryable operation which had been performed by the primary.) If the failure occurs between the call to

READUPDATE and the call to CHECKPOINT, the backup is reexecuting an operation already replied to. If the failure occurs between the call to CHECKPOINT and the call to REPLY, the backup completes the operation associated with the current request. In either case, the use of the "sync block" by the file system ensures that the request is not duplicated. Note also that the backup executes the call to REPLY at the end of the processing loop. This call is rejected, because there is no outstanding request on the backup side at this time (the rejection is ignored).

If the primary server process was in the midst of processing a request when a failure occurred, the backup server process receives the same request when it takes over. If the primary had not reached the checkpoint for the current operation, then the backup has the value of the preceding nonretryable sync ID. The backup sees the request as a new request and processes it. If the primary was processing a nonretryable request and had reached the checkpoint for the current operation, then the current sync ID was checkpointed (and, in fact, the backup completed the operation on its takeover). The backup sees this request as a duplicate request and returns the saved error code for the completed operation.

PROCESS REQUESTER DIRECTORY CHECKPOINT: This checkpoint is made, following the processing of a system message, in the "analyze^system^message" procedure. It checkpoints the entire requester process directory and the entire sync count table. This point was chosen for the directory checkpoint because directory changes may occur when system messages are received. (A directory change is also made in the "analyze^checkpoint^ status" procedure if the takeover occurred because of a processor module failure. However, a checkpoint at this point would be useless.)

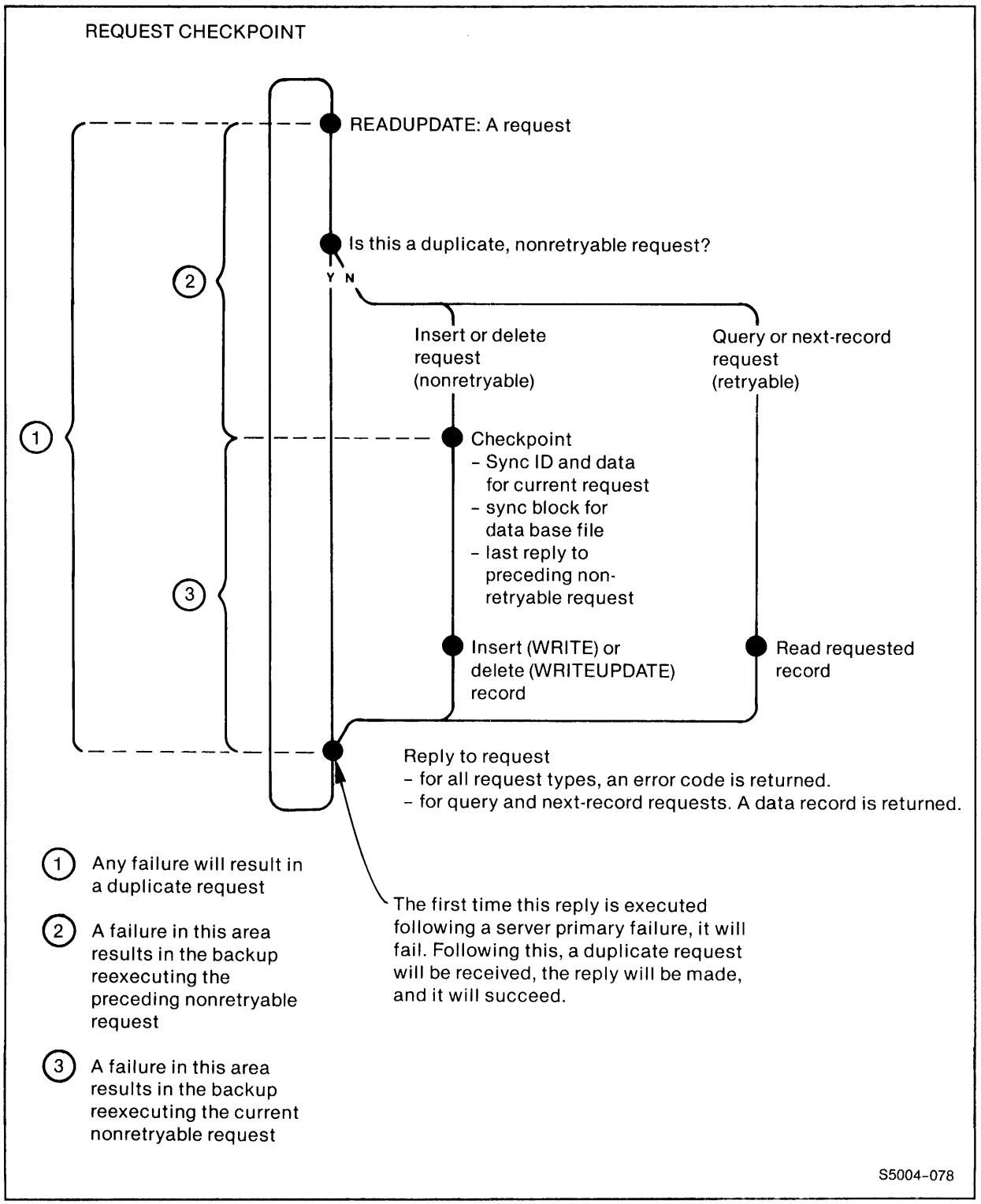


Figure B-2. Request Checkpoint

```
PAGE 1    $BOOKS1.M096B01.SERVESRC [1]

TAL - T9200D03 - (01JUL81)                          Source language: TAL - Target machine: Tandem NonStop System
Date - Time :  2/23/82 - 13:43:07                   Default options: On (LIST,CODE,MAP,WARN,LMAP) - Off (ICODE,INNERLIST)

    1.    000000 0  0  ! NonStop Programming Example
    2.    000000 0  0  ! for GUARDIAN
    3.    000000 0  0
    4.    000000 0  0
    5.    000000 0  0  ! Server Program
    6.    000000 0  0
    7.    000000 0  0  ?NOLIST
   10.    000000 0  0
   12.    000000 0  0
   13.    000000 0  0  ?LMAP*
   14.    000000 0  0
   15.    000000 0  0  ! This program is run by the following RUN command:
   16.    000000 0  0  !
   17.    000000 0  0  !    RUN serveobj / IN <data base file> , NAME $SERVE /
   18.    000000 0  0  !    --- -------- - -- ---------------- - ---- ------- -
   19.    000000 0  0  !
   20.    000000 0  0  !    where
   21.    000000 0  0  !
   22.    000000 0  0  !       <data base file> is the file name of the disc file to ba accessed.
   23.    000000 0  0  !
   24.    000000 0  0  !       $SERVE is the name used by requestors to access this process.
   25.    000000 0  0  !
   26.    000000 0  0  ! process NonStop state.
   27.    000000 0  0  INT backup^cpu,       ! backup cpu number.              ** NOT CHECKPOINTED.
   28.    000000 0  0      backup^pid[0:3],  ! process id of backup process.   ** NOT CHECKPOINTED.
   29.    000000 0  0      stop^count := 0,  ! to detect looping backup delete-creates
   30.    000000 0  0      backup^up :=  0,  ! true if backup is running.
   31.    000000 0  0     .stack^base;       ! beginning of data stack for checkpointing.
   32.    000000 0  0
   33.    000000 0  0  ! files
   34.    000000 0  0  INT recv^fnum,        ! $RECEIVE file number.
   35.    000000 0  0      db^fnum,          ! data base file number.
   36.    000000 0  0      db^fname[0:11];   ! data base file name.
   37.    000000 0  0
   38.    000000 0  0  LITERAL open^msgs       = %40000,
   39.    000000 0  0          protected       = %40,
   40.    000000 0  0          no^wait         = 1,
   41.    000000 0  0          recv^flags      = open^msgs + no^wait,
   42.    000000 0  0          recv^sync^depth = 1,
   43.    000000 0  0          db^flags        = protected,
   44.    000000 0  0          db^sync^depth   = 1;
   45.    000000 0  0
   46.    000000 0  0  ! data base format.
   47.    000000 0  0  LITERAL db^rec^len     = 256,
   48.    000000 0  0          db^rec^key^off =   0,
   49.    000000 0  0          db^rec^key^len =  24;
   50.    000000 0  0
   51.    000000 0  0  ! request message format.
   52.    000000 0  0  !
   53.    000000 0  0  !   word
   54.    000000 0  0  !   [0]     [1]             [2:message^size-3]
   55.    000000 0  0  !   [ sync ][ request type ][ -------------------- record -------------------- ]
   56.    000000 0  0  !
```

```
000000 0  0  !   sync = sync id ( incremented by requester on each request ).
000000 0  0  !          used to detect duplicate nonretryable requests.
000000 0  0  !
000000 0  0  !   request type.
000000 0  0  !
000000 0  0  !      0 = insert request       (nonretryable)
000000 0  0  !      1 = delete request.      (nonretryable)
000000 0  0  !      2 = query request.       (retryable)
000000 0  0  !      3 = next record request. (retryable)
000000 0  0  !
000000 0  0  !   record.
000000 0  0  !
000000 0  0  !      on request = record to be inserted, deleted, or obtained.
000000 0  0  !      on return for query or next = record from data base.
000000 0  0
000000 0  0  LITERAL
000000 0  0       sync              = 0,
000000 0  0       request^type      = 1,
000000 0  0       record            = 2,
000000 0  0       message^len       = db^rec^len + 4,
000000 0  0       message^size      = message^len / 2;
000000 0  0
000000 0  0  ! global buffers.
000000 0  0  INT .recv^buf[ 0 : message^size ],   ! receive buffer.
000203 0  0       recv^cnt,                       ! receive count.
000203 0  0      .buf     [ 0 : db^rec^len / 2 ]; ! scratch buffer.
000404 0  0
000404 0  0  LITERAL
000404 0  0       max^reqstrs = 16; ! maximum number of requestors allowed.
000404 0  0
000404 0  0  INT
000404 0  0       ! directory entry no. of previous requestor of nonretryable operation.
000404 0  0       old^requestor,
000404 0  0       ! sync id for latest requestor of nonretryable operation.
000404 0  0      .sync^count  [ 1 : max^reqstrs ] := max^reqstrs * [ 0 ],
000424 0  0       ! reply error code for each requestor.
000424 0  0      .reply^error [ 1 : max^reqstrs ],
000444 0  0       ! reply for current request.
000444 0  0      .reply^buf   [ 0 : message^size - 1 ];
000646 0  0
000646 0  0  ! requestor process directory.
000646 0  0  INT .pids[5:max^reqstrs * 5 + 5] := (max^reqstrs * 5) * [ 0 ];
000767 0  0
000767 0  0  !   [0]        [3] [4]
000767 0  0  !
000767 0  0  !   ┌──────────┬───┬───┐  entry #1
000767 0  0  !   ├──────────┼───┼───┤
000767 0  0  !   │          │   │   │  entry #2
000767 0  0  !   ├──────────┼───┼───┤
000767 0  0  !   │          │   │   │  entry #3
000767 0  0  !   ├──────────┼───┼───┤     .
000767 0  0  !   ~          ~   ~   ~     .
000767 0  0  !                           entry #max^reqstrs.
000767 0  0  !
000767 0  0  !
000767 0  0  !   entry^no[0:2] = <process name> or <creation time stamp>
000767 0  0  !   entry^no[3]   = <cpu,pin> of primary process
```

```
114.    000767 0  0  !   entry^no[4]    = <cpu,pin> of backup process, if any, or zero
115.    000767 0  0
116.    000767 0  0  ?NOLIST
```

```
PAGE 4    $BOOKS1.M096B01.SERVESRC [1] SERVE: read^start^up^message Procedure                    2/23/82   13:

  177.    000000 0  0  ! this procedure opens the $RECEIVE file for the primary process and reads
  178.    000000 0  0  ! the start-up message.  Note that the receipt of the start-up message
  179.    000000 0  0  ! involves reading three interprocess messages:
  180.    000000 0  0  !
  181.    000000 0  0  !  1.  OPEN "system" message.
  182.    000000 0  0  !  2.  CI start-up message (not a "system" message)
  183.    000000 0  0  !  3.  CLOSE "system" message.
  184.    000000 0  0
  185.    000000 0  0  PROC read^start^up^message;
  186.    000000 1  0
  187.    000000 1  0    BEGIN
  188.    000000 1  1      INT .recv^buf[0:33];
  189.    000000 1  1
  190.    000000 1  1      ! open $RECEIVE.
  191.    000000 1  1      recv^buf ':=' ["$RECEIVE", 8 * [ "  " ]];
  192.    000011 1  1      CALL OPEN ( recv^buf, recv^fnum, recv^flags, recv^sync^depth );
  193.    000023 1  1      IF <> THEN CALL ABEND;
  194.    000027 1  1
  195.    000027 1  1      ! read open message.
  196.    000027 1  1      CALL READ ( recv^fnum, recv^buf, 14 );
  197.    000037 1  1      CALL AWAITIO ( recv^fnum,,,, 3000D );
  198.    000050 1  1      IF <= OR recv^buf <> -30 THEN CALL ABEND;
  199.    000057 1  1
  200.    000057 1  1      ! read start^up message.
  201.    000057 1  1      CALL READ ( recv^fnum, recv^buf, 66 );
  202.    000067 1  1      CALL AWAITIO ( recv^fnum,,,, 3000D );
  203.    000100 1  1      IF <> OR recv^buf <> -1 THEN CALL ABEND;
  204.    000107 1  1
  205.    000107 1  1      ! save data^base file name.
  206.    000107 1  1      db^fname ':=' recv^buf[9] FOR 12;
  207.    000114 1  1
  208.    000114 1  1      ! read close message.
  209.    000114 1  1      CALL READ ( recv^fnum, recv^buf, 14 );
  210.    000124 1  1      CALL AWAITIO ( recv^fnum,,,, 3000D );
  211.    000135 1  1      IF <= OR recv^buf <> -31 THEN CALL ABEND;
  212.    000144 1  1    END; ! read^start^up^message.

RECV^BUF                          Variable        INT        L+001    Indirect

00000  070402 024700 002042 170401 000025 020137 000200 100014     00010 026047 170401 070010 005100 004001 100001 024733 002
00020  100360 024700 027000 012003 000002 024711 027000 040010     00030 170401 100016 024722 002003 100034 024700 027000 070
00040  024700 002003 100000 005013 004270 100021 024722 027000     00050 016003 140401 001742 012003 000002 024711 027000 040
00060  170401 100102 024722 002003 100034 024700 027000 070010     00070 024700 002003 100000 005013 004270 100021 024722 027
00100  015003 140401 001777 012003 000002 024711 027000 070012     00110 103011 173401 100014 026007 040010 170401 100016 024
00120  002003 100034 024700 027000 070010 024700 002003 100000     00130 005013 004270 100021 024722 027000 016003 140401 001
00140  012003 000002 024711 027000 125003 000141 022122 042503     00150 042511 053105 020040 020040 020040 020040 020040 020
00160  020040 020040
```

```
PAGE 5    $BOOKS1.M096B01.SERVESRC [1] SERVE: File Open Procedures                    2/23/82   13:

  214.    000000 0  0  ! this procedure opens the data base file for the primary process.
  215.    000000 0  0
  216.    000000 0  0  PROC open^primarys^files;
  217.    000000 1  0
  218.    000000 1  0    BEGIN
  219.    000000 1  1
  220.    000000 1  1      ! open data base file.
  221.    000000 1  1      CALL OPEN ( db^fname, db^fnum, db^flags, db^sync^depth );
  222.    000011 1  1      IF <> THEN CALL ABEND;
  223.    000015 1  1    END; ! open^primarys^files.


00000  070012 070011 100040 100001 024733 002004 100360 024700     00010 027000 012003 000002 024711 027000 125003

  224.    000000 0  0  ! this procedure opens the $RECEIVE and data base files for the backup
  225.    000000 0  0  ! process.
  226.    000000 0  0
  227.    000000 0  0  PROC open^backups^files;
  228.    000000 1  0
  229.    000000 1  0    BEGIN
  230.    000000 1  1      INT .buf[0:11],
  231.    000000 1  1          back^error;
  232.    000000 1  1
  233.    000000 1  1      ! open $RECEIVE.
  234.    000000 1  1      buf ':=' ["$RECEIVE", 8 * [ "  " ]];
  235.    000011 1  1      CALL CHECKOPEN ( buf, recv^fnum, recv^flags, recv^sync^depth,,,
  236.    000011 1  1                       back^error );
  237.    000024 1  1      IF <> THEN CALL STOP ( backup^pid );
  238.    000031 1  1
  239.    000031 1  1      ! open data base file.
  240.    000031 1  1      CALL CHECKOPEN ( db^fname, db^fnum, db^flags, db^sync^depth,,,
  241.    000031 1  1                       back^error );
  242.    000043 1  1      IF <> THEN CALL STOP ( backup^pid );
  243.    000050 1  1    END; ! open^backups^files.

BACK^ERROR                      Variable        INT       L+002    Direct
BUF                             Variable        INT       L+001    Indirect


00000  070403 024700 002015 170401 000025 020043 000200 100014     00010 026047 170401 040010 005100 004001 100001 000002 070
00020  024766 100171 024700 027000 012004 070001 100001 024711     00030 027000 070012 040011 100040 100001 000002 070402 024
00040  100171 024700 027000 012004 070001 100001 024711 027000     00050 125003 000045 022122 042503 042511 053105 020040 020
00060  020040 020040 020040 020040 020040 020040
```

```
PAGE 6    $BOOKS1.M096B01.SERVESRC [1] SERVE: create^backup Procedure                    2/23/82   13:

  245.    000000 0  0  ! this procedure creates the backup process.  The steps involved are:
  246.    000000 0  0  !
  247.    000000 0  0  ! - create backup.
  248.    000000 0  0  ! - open its files.
  249.    000000 0  0  ! - checkpoint current state.
  250.    000000 0  0
  251.    000000 0  0  PROC create^backup ( backup^cpu );
  252.    000000 1  0     INT backup^cpu;
  253.    000000 1  0
  254.    000000 1  0    BEGIN
  255.    000000 1  1      INT .pfile[0:11],
  256.    000000 1  1          pname[0:3],
  257.    000000 1  1          error,
  258.    000000 1  1          status,
  259.    000000 1  1         .globals := 5; ! base for initial checkpoint = 'G'[5].  "backup^cpu"
  260.    000000 1  1                        ! and "backup^pid" not checkpointed.
  261.    000000 1  1
  262.    000000 1  1      ! check for looping creates.
  263.    000000 1  1      IF stop^count > 5 THEN CALL DEBUG;
  264.    000012 1  1
  265.    000012 1  1      ! get program's file name.
  266.    000012 1  1      CALL PROGRAMFILENAME ( pfile );
  267.    000015 1  1      ! get process's name.
  268.    000015 1  1      CALL GETCRTPID ( MYPID, pname );
  269.    000021 1  1      ! create the backup process.
  270.    000021 1  1      CALL NEWPROCESS ( pfile,,, backup^cpu, backup^pid, error, pname );
  271.    000033 1  1      IF backup^pid THEN ! it was created.
  272.    000035 1  1        BEGIN
  273.    000035 1  2          backup^up := 1;
  274.    000037 1  2          CALL open^backups^files;
  275.    000040 1  2
  276.    000040 1  2          ! checkpoint global area through top-of-stack and sync block.
  277.    000040 1  2          IF ( status := CHECKPOINT ( globals, ,db^fnum ) ) THEN          ! *** CHECKPOINT *** !
  278.    000053 1  2            CALL analyze^checkpoint^status ( status );
  279.    000056 1  2        END;
  280.    000056 1  1    END; ! of create^backup

BACKUP^CPU                      Variable        INT       L-003    Direct
ERROR                           Variable        INT       L+006    Direct
GLOBALS                         Variable        INT       L+010    Indirect
PFILE                           Variable        INT       L+001    Indirect
PNAME                           Variable        INT       L+002    Direct
STATUS                          Variable        INT       L+007    Direct


00000  070411 024700 002006 100005 024700 002014 040005 001005     00010 016001 027000 170401 024700 027000 027000 070402 024
00020  027000 170401 000002 040703 070001 070406 024755 070402     00030 100117 024711 027000 040001 014421 100001 044006 027
00040  170410 100000 040011 024722 002030 005005 100000 024711     00050 027000 034407 014403 040407 024700 027000 125004
```

```
  282.    000000 0  0  ! this procedure is used to analyze and take appropriate action for a
  283.    000000 0  0  ! non-zero return from the CHECKPOINT procedure.
  284.    000000 0  0
  285.    000000 0  0  PROC analyze^checkpoint^status ( status );
  286.    000000 1  0     INT status; ! return value of CHECKPOINT.
  287.    000000 1  0
  288.    000000 1  0    BEGIN
  289.    000000 1  1
  290.    000000 1  1      IF backup^up THEN ! analyze^it.
  291.    000002 1  1        CASE status.<0:7> OF
  292.    000005 1  1          BEGIN
  293.    000005 1  2  ! 0 !     ;     ! good checkpoint.
  294.    000005 1  2
  295.    000005 1  2  ! 1 !     BEGIN ! checkpoint failure.
  296.    000005 1  3              ! find out if backup is still running.
  297.    000005 1  3              CALL GETCRTPID ( backup^pid[3], backup^pid );
  298.    000011 1  3              IF = THEN ! backup still running.
  299.    000012 1  3                BEGIN
  300.    000012 1  4                  ! stop the backup.
  301.    000012 1  4                  CALL STOP ( backup^pid );
  302.    000016 1  4                  backup^up := 0;
  303.    000020 1  4                END;
  304.    000020 1  3            END;  ! 1.
  305.    000021 1  2
  306.    000021 1  2  ! 2 !     BEGIN ! takeover from primary.
  307.    000021 1  3              CASE status.<8:15> OF
  308.    000024 1  3                BEGIN
  309.    000024 1  4
  310.    000024 1  4  ! 2-0 !         BEGIN ! primary stopped, so stop myself.
  311.    000024 1  5                    CALL STOP;
  312.    000027 1  5                  END;
  313.    000030 1  4
  314.    000030 1  4  ! 2-1 !         BEGIN ! primary abended, so create a backup for me.
  315.    000030 1  5                    backup^up := 0;
  316.    000032 1  5                    stop^count := stop^count + 1;
  317.    000034 1  5                    CALL create^backup ( backup^cpu );
  318.    000037 1  5                  END;
  319.    000040 1  4
  320.    000040 1  4  ! 2-2 !         BEGIN ! primary cpu down, note it.
  321.    000040 1  5                    backup^up := 0;
  322.    000042 1  5                    CALL delallpids ( backup^cpu );
  323.    000045 1  5                  END;
  324.    000046 1  4
  325.    000046 1  4  ! 2-3 !         ! primary called CHECKSWITCH.
  326.    000046 1  4                  ;
  327.    000046 1  4                END; ! case of status.<8:15>.
  328.    000053 1  3            END;  ! 2.
  329.    000054 1  2
  330.    000054 1  2  ! 3 !     BEGIN ! bad parameter to CHECKPOINT
  331.    000054 1  3              CALL DEBUG;
  332.    000055 1  3            END;  ! 3.
  333.    000056 1  2
  334.    000056 1  2          END; ! case of status.<0:7>.
  335.    000063 1  1    END; ! analyze^checkpoint^status.

STATUS                        Variable        INT        L-003    Direct
```

```
00000  040006 014461 040703 030110 010451 040004 070001 024711     00010 027000 015006 070001 100001 024711 027000 100000 044
00020  010442 040703 006377 010422 000002 024711 027000 010423     00030 100000 044006 100001 074005 040000 024700 027000 010
00040  100000 044006 040000 024700 027000 010405 000030 177755     00050 177760 177767 000001 010407 027000 010405 000030 000
00060  177725 177740 177772 125004
```

```
PAGE 9    $BOOKS1.M096B01.SERVESRC [1] SERVE: lookuppid Procedure                    2/23/82   13:

  337.    000000 0  0  ! This procedure searches the requestor process directory by a process id
  338.    000000 0  0  ! for an entry number.
  339.    000000 0  0  !
  340.    000000 0  0  !   return values:
  341.    000000 0  0  !     0 = pid not in directory.
  342.    000000 0  0  !    >0 = entry no of pid in directory.
  343.    000000 0  0
  344.    000000 0  0  INT PROC lookuppid(pid);
  345.    000000 1  0     INT .pid;
  346.    000000 1  0
  347.    000000 1  0    BEGIN
  348.    000000 1  1      INT entry^no := 0, ! entry^no in local pid directory.
  349.    000000 1  1          comp^len;      ! compare length for pid matching.
  350.    000000 1  1
  351.    000000 1  1      comp^len := IF pid.<0:7> = "$" THEN ! process name ! 3 ELSE 4;
  352.    000013 1  1      WHILE (entry^no := entry^no + 1) <= max^reqstrs DO
  353.    000020 1  1        IF pid = pids[entry^no * 5] FOR comp^len THEN ! found it.
  354.    000031 1  1          RETURN entry^no;
  355.    000034 1  1
  356.    000034 1  1      RETURN 0; ! not found.
  357.    000036 1  1    END; ! lookuppid.

COMP^LEN                        Variable        INT        L+002    Direct
ENTRY^NO                        Variable        INT        L+001    Direct
PID                             Variable        INT        L-003    Indirect


00000  100000 024700 002001 140703 030110 001044 015002 100003     00010 010401 100004 044402 040401 104001 034401 001020 011
00020  170703 040401 100005 000212 000117 173035 040402 026207     00030 015002 040401 125004 010757 100000 125004
```

```
PAGE 10   $BOOKS1.M096B01.SERVESRC [1] SERVE: addpid Procedure                2/23/82   13:

  359.    000000 0  0  ! This procedure adds a process id to the requestor process directory.
  360.    000000 0  0  !
  361.    000000 0  0  !   return values.
  362.    000000 0  0  !     0 = directory full, "pid" not added to directory.
  363.    000000 0  0  !    >0 = "pid" added, entry no of "pid" in directory.
  364.    000000 0  0
  365.    000000 0  0  INT PROC addpid(pid);
  366.    000000 1  0     INT .pid;
  367.    000000 1  0
  368.    000000 1  0    BEGIN
  369.    000000 1  1      INT entry^no,                  ! entry^no in local pid directory.
  370.    000000 1  1          zero[0:3] := [ 4 * [ 0 ]]; ! for lookup of empty directory slot.
  371.    000004 1  1
  372.    000004 1  1      IF (entry^no := lookuppid(pid)) THEN  ! already in directory.
  373.    000017 1  1        BEGIN
  374.    000017 1  2          ! check for duplicate open.
  375.    000017 1  2          IF pids[ entry^no * 5 + 3 ] <> pid[ 3 ] AND
  376.    000017 1  2             pids[ entry^no * 5 + 4 ] <> pid[ 3 ] THEN ! first open by
  377.    000042 1  2             pids[ entry^no * 5 + 4 ] := pid[ 3 ];     ! backup.
  378.    000044 1  2        END
  379.    000044 1  1      ELSE ! not in directory.  First open by "pid"
  380.    000045 1  1        BEGIN
  381.    000045 1  2          IF (entry^no := lookuppid(zero)) THEN ! look for empty slot.
  382.    000052 1  2            BEGIN
  383.    000052 1  3              pids[entry^no * 5] ':=' pid FOR 4;
  384.    000062 1  3              sync^count[entry^no] := -1; ! initialize requestor id^count.
  385.    000065 1  3            END;
  386.    000065 1  2        END;
  387.    000065 1  1      RETURN entry^no;   ! this returns zero if no room in directory.
  388.    000067 1  1    END; ! addpid.

ENTRY^NO                        Variable        INT       L+001    Direct
PID                             Variable        INT       L-003    Indirect
ZERO                            Variable        INT       L+002    Direct

00000  000000 000000 000000 000000 002005 070402 000025 003771     00010 100004 026047 170703 024700 027000 034401 014426 040
00020  100005 000212 104003 000117 143035 102003 142703 000215     00030 012013 040401 100005 000212 104004 000115 141035 142
00040  000215 012002 142703 145035 010420 070402 024700 027000     00050 034401 014413 040401 100005 000212 000117 173035 170
00060  100004 026007 032401 100777 146032 040401 125004
```

```
PAGE 11   $BOOKS1.M096B01.SERVESRC [1] SERVE: delpid Procedure                    2/23/82   13:

  390.    000000 0  0  ! This procedure deletes a process id from the requestor process
  391.    000000 0  0  ! directory.
  392.    000000 0  0
  393.    000000 0  0  PROC delpid(pid);
  394.    000000 1  0    INT .pid; ! "pid" to be deleted.
  395.    000000 1  0
  396.    000000 1  0    BEGIN
  397.    000000 1  1      INT entry^no; ! entry^no in local pid directory.
  398.    000000 1  1
  399.    000000 1  1      IF (entry^no := lookuppid(pid)) THEN  ! delete it.
  400.    000006 1  1        IF pids[entry^no * 5 + 4] THEN ! was open by process-pair.
  401.    000015 1  1          BEGIN
  402.    000015 1  2            IF pids[entry^no * 5 + 3] = pid[3] THEN ! close by primary.
  403.    000027 1  2              ! replace primary entry^no with backup.
  404.    000027 1  2              pids[entry^no * 5 + 3] := pids[entry^no * 5 + 4];
  405.    000031 1  2            ! clear backup entry^no.
  406.    000031 1  2            pids[entry^no * 5 + 4] := 0;
  407.    000040 1  2          END
  408.    000040 1  1        ELSE ! was open by one process.
  409.    000041 1  1          pids[entry^no * 5] ':=' [0,0,0,0];
  410.    000053 1  1    END; ! delpid.

ENTRY^NO                          Variable        INT        L+001    Direct
PID                               Variable        INT        L-003    Indirect


00000  002001 170703 024700 027000 034401 014445 040401 100005     00010 000212 104004 000117 143035 014424 040401 100005 000
00020  104003 000116 142035 101003 141703 000215 015002 143035     00030 146035 040401 100005 000212 104004 000117 100000 147
00040  010412 040401 100005 000212 000117 173035 000025 020004     00050 000200 100004 026047 125004 000006 000000 000000 000
00060  000000
```

```
PAGE 12   $BOOKS1.M096B01.SERVESRC [1] SERVE: delallpids Procedure                2/23/82   13:

  412.    000000 0  0  ! this procedure is called if a cpu failure message is received.  It
  413.    000000 0  0  ! deletes all references in the requestor process directory to the
  414.    000000 0  0  ! failed cpu.  This may cause entire entries to be deleted.
  415.    000000 0  0
  416.    000000 0  0  PROC delallpids(cpu);
  417.    000000 1  0     INT cpu; ! processor module number of pids to be deleted.
  418.    000000 1  0
  419.    000000 1  0    BEGIN
  420.    000000 1  1      INT entry^no := 0,   ! entry^no in local pid directory.
  421.    000000 1  1          temp;
  422.    000000 1  1
  423.    000000 1  1      WHILE (entry^no := entry^no + 1) <= max^reqstrs DO
  424.    000010 1  1        BEGIN ! check each entry^no.
  425.    000010 1  2
  426.    000010 1  2          ! check for match with entry^no's primary cpu.
  427.    000010 1  2          IF pids[ entry^no * 5 + 3 ] AND
  428.    000010 1  2             pids[ entry^no * 5 + 3 ].<0:7> = cpu THEN ! primary down
  429.    000024 1  2            ! delete primary process and maybe the entire entry^no.
  430.    000024 1  2            CALL delpid ( pids [ entry^no * 5 ] )
  431.    000030 1  2          ELSE
  432.    000031 1  2          ! check for match with entry^no's backup cpu.
  433.    000031 1  2          IF pids[ entry^no * 5 + 4 ] AND
  434.    000031 1  2             pids[ entry^no * 5 + 4 ].<0:7> = cpu THEN ! backup down.
  435.    000045 1  2            ! clear the backup entry^no.
  436.    000045 1  2            pids[ entry^no * 5 + 4 ] := 0;
  437.    000047 1  2        END;
  438.    000050 1  1    END; ! delallpids.

CPU                             Variable        INT        L-003    Direct
ENTRY^NO                        Variable        INT        L+001    Direct
TEMP                            Variable        INT        L+002    Direct

00000  100000 024700 002001 040401 104001 034401 001020 011040     00010 040401 100005 000212 104003 000117 143035 014412 143
00020  030110 040703 000215 015005 107775 173035 024700 027000     00030 010416 040401 100005 000212 104004 000117 143035 014
00040  143035 030110 040703 000215 015002 100000 147035 010733     00050 125004
```

```
440.    000000 0  0  ! this procedure analyzes and takes appropriate action for system messages.
441.    000000 0  0
442.    000000 0  0  PROC analyze^system^message ( recv^buf, recv^cnt, pid );
443.    000000 1  0     INT .recv^buf,
444.    000000 1  0         recv^cnt,
445.    000000 1  0        .pid;
446.    000000 1  0
447.    000000 1  0    BEGIN
448.    000000 1  1      INT reply^error := 0,
449.    000000 1  1          status;
450.    000000 1  1
451.    000000 1  1      CASE $ABS ( recv^buf ) OF
452.    000007 1  1        BEGIN
453.    000007 1  2  !  0  ! ;
454.    000007 1  2  !  1  ! ;
455.    000007 1  2
456.    000007 1  2  !  2  ! BEGIN ! cpu down.
457.    000007 1  3           ! delete any references in the requestor process
458.    000007 1  3           ! directory to the cpu that failed.
459.    000007 1  3           CALL delallpids ( recv^buf[1] );
460.    000013 1  3           IF recv^buf[1] = backup^cpu THEN backup^up := 0;
461.    000022 1  3         END;
462.    000023 1  2
463.    000023 1  2  !  3  ! BEGIN ! cpu up.
464.    000023 1  3           IF recv^buf[1] = backup^cpu THEN ! backup cpu came up.
465.    000030 1  3             BEGIN
466.    000030 1  4               ! clear stop count.
467.    000030 1  4               stop^count := 0;
468.    000032 1  4               CALL create^backup ( backup^cpu );
469.    000035 1  4             END;
470.    000035 1  3         END;
471.    000036 1  2
472.    000036 1  2  !  4  ! ;
473.    000036 1  2
474.    000036 1  2  !  5  ! BEGIN ! backup stopped.
475.    000036 1  3           backup^up := 0;
476.    000040 1  3           stop^count := stop^count + 1;
477.    000042 1  3           CALL create^backup ( backup^cpu );
478.    000045 1  3         END;
479.    000046 1  2
480.    000046 1  2  !  6  ! BEGIN ! backup abended.
481.    000046 1  3           backup^up := 0;
482.    000050 1  3           stop^count := stop^count + 1;
483.    000052 1  3           CALL create^backup ( backup^cpu );
484.    000055 1  3         END;
485.    000056 1  2
486.    000056 1  2  ! 7-29! ;;;;;;;;;;;;;;;;;;;;;;;
487.    000056 1  2
488.    000056 1  2  !  30 ! BEGIN ! OPEN system message.
489.    000056 1  3           ! check for no-wait i/o depth > 1.
490.    000056 1  3           IF recv^buf[ 1 ].<12:15> > 1 THEN
491.    000063 1  3             reply^error := 28  ! return illegal no-wait depth error.
492.    000063 1  3           ELSE
493.    000066 1  3           ! try to add opener to directory.
494.    000066 1  3           IF NOT addpid ( pid ) THEN
495.    000072 1  3             reply^error := 12; ! return file in use error.
496.    000074 1  3         END;
```

```
PAGE 14   $BOOKS1.M096B01.SERVESRC [1] SERVE: analyze^system^message Procedure                    2/23/82   13:

  497.    000075 1  2
  498.    000075 1  2  !  31 ! BEGIN ! CLOSE system message.
  499.    000075 1  3           ! delete closer from requestor process directory.
  500.    000075 1  3           CALL delpid ( pid );
  501.    000100 1  3         END;
  502.    000101 1  2
  503.    000101 1  2         OTHERWISE reply^error := 2;    ! return invalid operation error.
  504.    000103 1  2       END; ! system message case.
  505.    000154 1  1
  506.    000154 1  1     IF ( status := CHECKPOINT ( stack^base,                          ! *** CHECKPOINT *** !
  507.    000154 1  1                                 pids [ 5 ], max^reqstrs * 5 + 5,
  508.    000154 1  1                                 sync^count [ 1 ], max^reqstrs,
  509.    000154 1  1                                 stop^count, 1,
  510.    000154 1  1                                 reply^error [ old^requestor ], 1,
  511.    000154 1  1                                 ,db^fnum) ) THEN
  512.    000204 1  1        CALL analyze^checkpoint^status ( status );
  513.    000207 1  1
  514.    000207 1  1     ! reply to system message.
  515.    000207 1  1     CALL REPLY (,,,, reply^error);
  516.    000214 1  1   END; ! analyze^system^message.

PID                             Variable        INT        L-003    Indirect
RECV^BUF                        Variable        INT        L-005    Indirect
RECV^CNT                        Variable        INT        L-004    Direct
REPLY^ERROR                     Variable        INT        L+001    Direct
STATUS                          Variable        INT        L+002    Direct

00000  100000 024700 002001 140705 013001 000214 010475 103001     00010 143705 024700 027000 103001 143705 040000 000215 015
00020  100000 044006 010531 103001 143705 040000 000215 015005     00030 100000 044005 040000 024700 027000 010516 100000 044
00040  100001 074005 040000 024700 027000 010506 100000 044006     00050 100001 074005 040000 024700 027000 010476 103001 143
00060  006017 001001 016003 100034 044401 010406 170703 024700     00070 027000 015402 100014 044401 010457 170703 024700 027
00100  010453 100002 044401 010450 100037 000205 011002 000100     00110 010401 100040 000030 000041 000040 177672 177705 000
00120  177716 177725 000032 000031 000030 000027 000026 000025     00130 000024 000023 000022 000021 000020 000017 000016 000
00140  000014 000013 000012 000011 000010 000007 000006 000005     00150 000004 177705 177723 177726 170007 103005 173035 100
00160  102001 172032 100020 070005 024755 100001 031031 071401     00170 100001 100000 040011 024744 002020 005007 004375 100
00200  024711 027000 034402 014403 040402 024700 027000 002004     00210 040401 100001 024711 027000 125006
```

```
PAGE 15   $BOOKS1.M096B01.SERVESRC [1] SERVE: query^request Procedure                    2/23/82   13:

  518.    000000 0  0  ! this procedure searces the data base for the record associated with
  519.    000000 0  0  ! a key value.
  520.    000000 0  0  !
  521.    000000 0  0  ! return values.
  522.    000000 0  0  !  0 = record found, record in "result".
  523.    000000 0  0  ! >0 = file management error.
  524.    000000 0  0
  525.    000000 0  0  INT PROC query^request ( rec, result, result^len );
  526.    000000 1  0
  527.    000000 1  0     INT .rec,        ! key value.
  528.    000000 1  0         .result,     ! record of key.
  529.    000000 1  0         .result^len; ! length of record of key.
  530.    000000 1  0
  531.    000000 1  0    BEGIN
  532.    000000 1  1      INT error;
  533.    000000 1  1
  534.    000000 1  1      result^len := 0;
  535.    000003 1  1      CALL KEYPOSITION ( db^fnum, rec,,,2 );
  536.    000013 1  1      CALL READUPDATE ( db^fnum, result, db^rec^len, result^len );
  537.    000024 1  1      CALL FILEINFO ( db^fnum, error );
  538.    000033 1  1      RETURN error;
  539.    000035 1  1    END; ! query^request.

ERROR                          Variable        INT        L+001    Direct
REC                            Variable        INT        L-005    Indirect
RESULT                         Variable        INT        L-004    Indirect
RESULT^LEN                     Variable        INT        L-003    Indirect

00000  002001 100000 144703 040011 170705 030001 000002 100002     00010 100031 024755 027000 040011 170704 005001 170703 000
00020  024755 100036 024700 027000 040011 070401 024711 002013     00030 005030 024700 027000 040401 125006
```

```
PAGE 16   $BOOKS1.M096B01.SERVESRC [1] SERVE: insert^request Procedure                2/23/82   13:

  541.    000000 0  0  ! this procedure adds a record to the data base.
  542.    000000 0  0  !
  543.    000000 0  0  ! return values.
  544.    000000 0  0  !
  545.    000000 0  0  !  0 = record "rec" added.
  546.    000000 0  0  ! >0 = file management error.
  547.    000000 0  0
  548.    000000 0  0  INT PROC insert^request ( rec );
  549.    000000 1  0     INT .rec;     ! record to be added.
  550.    000000 1  0
  551.    000000 1  0    BEGIN
  552.    000000 1  1
  553.    000000 1  1      INT  error;
  554.    000000 1  1
  555.    000000 1  1      ! insert the record.
  556.    000000 1  1      CALL WRITE ( db^fnum, rec, db^rec^len );
  557.    000011 1  1      CALL FILEINFO ( db^fnum, error );
  558.    000020 1  1      RETURN error;
  559.    000022 1  1    END; ! insert^request.

ERROR                          Variable        INT        L+001    Direct
REC                            Variable        INT        L-003    Indirect

00000  002001 040011 170703 005001 024722 002003 100034 024700     00010 027000 040011 070401 024711 002013 005030 024700 027
00020  040401 125004
```

```
PAGE 17   $BOOKS1.M096B01.SERVESRC [1] SERVE: delete^request Procedure                2/23/82   13:

  561.    000000 0  0  ! this procedure deletes a record from the data base file.
  562.    000000 0  0  !
  563.    000000 0  0  ! return values.
  564.    000000 0  0  !
  565.    000000 0  0  !   0 = record "rec" deleted.
  566.    000000 0  0  !  >0 = file management error.
  567.    000000 0  0
  568.    000000 0  0  INT PROC delete^request ( rec );
  569.    000000 1  0     INT .rec;     ! key of record to be deleted.
  570.    000000 1  0
  571.    000000 1  0    BEGIN
  572.    000000 1  1      INT  error;
  573.    000000 1  1
  574.    000000 1  1      ! delete the record.
  575.    000000 1  1      CALL KEYPOSITION ( db^fnum, rec );
  576.    000011 1  1      CALL WRITEUPDATE ( db^fnum, rec, 0 );
  577.    000021 1  1      CALL FILEINFO ( db^fnum, error );
  578.    000030 1  1      RETURN error;
  579.    000032 1  1    END; ! delete^request.

ERROR                           Variable        INT        L+001    Direct
REC                             Variable        INT        L-003    Indirect

00000  002001 040011 170703 030001 024711 002003 100030 024700     00010 027000 040011 170703 100000 024722 002003 100034 024
00020  027000 040011 070401 024711 002013 005030 024700 027000     00030 040401 125004
```

```
  581.    000000 0  0  ! this procedure returns the next record in the data base file.
  582.    000000 0  0  !
  583.    000000 0  0  ! calling values.
  584.    000000 0  0  !
  585.    000000 0  0  !   rec = 0, return first record in file.
  586.    000000 0  0  !   rec =  record, return next record in file.
  587.    000000 0  0  !
  588.    000000 0  0  ! return values.
  589.    000000 0  0  !
  590.    000000 0  0  !   0 = first/next record returned in "result".
  591.    000000 0  0  !  >0 = file management error.
  592.    000000 0  0
  593.    000000 0  0  INT PROC next^record^request ( rec, result, result^len );
  594.    000000 1  0    INT .rec,        ! key of record for positioning.
  595.    000000 1  0        .result,     ! next record.
  596.    000000 1  0        .result^len; ! length of next record.
  597.    000000 1  0
  598.    000000 1  0    BEGIN
  599.    000000 1  1
  600.    000000 1  1      INT error;
  601.    000000 1  1
  602.    000000 1  1      STRING
  603.    000000 1  1          .srec := @rec '<<' 1;
  604.    000000 1  1
  605.    000000 1  1      ! increment key value past current value.
  606.    000000 1  1      srec [ db^rec^key^off + db^rec^key^len - 1 ] :=
  607.    000004 1  1         srec [ db^rec^key^off + db^rec^key^len - 1 ] '+' 1;
  608.    000010 1  1
  609.    000010 1  1      CALL KEYPOSITION ( db^fnum, rec );
  610.    000020 1  1      CALL READ ( db^fnum, result, db^rec^len, result^len );
  611.    000031 1  1      CALL FILEINFO ( db^fnum, error );
  612     000040 1  1      RETURN error;
  613.    000042 1  1    END; ! next^record^request.

ERROR                          Variable        INT       L+001    Direct
REC                            Variable        INT       L-005    Indirect
RESULT                         Variable        INT       L-004    Indirect
RESULT^LEN                     Variable        INT       L-003    Indirect
SREC                           Variable        STRING    L+002    Indirect


00000  002001 170705 030001 024700 103027 153402 003001 157402     00010 040011 170705 030001 024711 002003 100030 024700 027
00020  040011 170704 005001 170703 000002 024755 100036 024700     00030 027000 040011 070401 024711 002013 005030 024700 027
00040  040401 125006
```

```
PAGE 19   $BOOKS1.M096B01.SERVESRC [1] SERVE: process^user^request Procedure                2/23/82   13:

  615.    000000 0  0  ! this procedure is used to process a request.
  616.    000000 0  0
  617.    000000 0  0  PROC process^user^request ( recv^buf, recv^cnt, pid );
  618.    000000 1  0     INT .recv^buf,
  619.    000000 1  0         recv^cnt,
  620.    000000 1  0        .pid; ! process id of requestor.
  621.    000000 1  0
  622.    000000 1  0    BEGIN
  623.    000000 1  1      INT  status,
  624.    000000 1  1           requestor,      ! directory entry no. of current requestor.
  625.    000000 1  1           reply^len := 0; ! reply length for current request.
  626.    000000 1  1
  627.    000000 1  1      ! get requestor number of current requestor.
  628.    000000 1  1      IF NOT ( requestor := lookuppid ( pid ) ) THEN ! invalid requestor.
  629.    000010 1  1        BEGIN
  630.    000010 1  2          CALL REPLY ( ,,,, 60 ); ! return "device has been downed" error.
  631.    000015 1  2          RETURN;
  632.    000016 1  2        END;
  633.    000016 1  1
  634.    000016 1  1      ! check for duplicate request.
  635.    000016 1  1      IF recv^buf [ sync ] <> sync^count[ requestor ] THEN ! not duplicate, nonretryable.
  636.    000023 1  1        BEGIN
  637.    000023 1  2
  638.    000023 1  2          IF recv^buf [ request^type ] <= 1 THEN ! nonretryable, so checkpoint current state to backup.
  639.    000027 1  2            BEGIN
  640.    000027 1  3              ! save sync count of current requestor.
  641.    000027 1  3              sync^count [ requestor ] := recv^buf;
  642.    000031 1  3
  643.    000031 1  3              IF ( status := CHECKPOINT ( stack^base,                    ! *** CHECKPOINT *** !
  644.    000031 1  3                                          ,db^fnum,
  645.    000031 1  3                                          sync^count [ requestor ], 1,
  646.    000031 1  3                                          reply^error [ old^requestor ], 1,
  647.    000031 1  3                                          recv^buf, ( recv^cnt + 1 ) / 2 ) ) THEN
  648.    000060 1  3                 CALL analyze^checkpoint^status ( status );
  649.    000063 1  3
  650.    000063 1  3              ! save requestor number of current requestor for a subsequent checkpoint.
  651.    000063 1  3              old^requestor := requestor;
  652.    000065 1  3            END; ! of checkpoint.
  653.    000065 1  2
  654.    000065 1  2          ! save request^type.
  655.    000065 1  2          reply^buf ':=' recv^buf FOR 2;
  656.    000071 1  2
  657.    000071 1  2          ! process the data base request.
  658.    000071 1  2          CASE recv^buf [ request^type ] OF
  659.    000074 1  2            BEGIN
  660.    000074 1  3
  661.    000074 1  3              ! "insert" request.
  662.    000074 1  3              reply^error [ requestor ] := insert^request ( recv^buf [ record ] );
  663.    000103 1  3
  664.    000103 1  3              ! "delete" request.
  665.    000103 1  3              reply^error [ requestor ] := delete^request ( recv^buf [ record ] );
  666.    000112 1  3
  667.    000112 1  3              ! "query" request.
  668.    000112 1  3              reply^error [ requestor ] :=
  669.    000112 1  3                 query^request ( recv^buf [ record ], reply^buf [ record ], reply^len );
  670.    000123 1  3
  671.    000123 1  3              ! "next entry" request.
```

```
PAGE 20   $BOOKS1.M096B01.SERVESRC [1] SERVE: process^user^request Procedure                    2/23/82   13:

  672.    000123 1  3              reply^error [ requestor ] :=
  673.    000123 1  3                 next^record^request ( recv^buf [ record ], reply^buf [ record ], reply^len );
  674.    000134 1  3
  675.    000134 1  3              OTHERWISE reply^error [ requestor ] := 29; ! bad param.
  676.    000137 1  3            END;
  677.    000154 1  2         END;
  678.    000154 1  1
  679.    000154 1  1       ! return the reply to the requestor.
  680.    000154 1  1       CALL REPLY ( reply^buf, reply^len + 4, , , reply^error [ requestor ] );
  681.    000165 1  1
  682.    000165 1  1     END; ! process^user^request.

PID                              Variable        INT        L-003    Indirect
RECV^BUF                         Variable        INT        L-005    Indirect
RECV^CNT                         Variable        INT        L-004    Direct
REPLY^LEN                        Variable        INT        L+003    Direct
REQUESTOR                        Variable        INT        L+002    Direct
STATUS                           Variable        INT        L+001    Direct

00000  002002 100000 024700 170703 024700 027000 034402 015406     00010 002004 100074 100001 024711 027000 125006 140705 033
00020  143032 000215 012131 102001 142705 001001 011036 140705     00030 147032 170007 100000 040011 173032 100001 024744 031
00040  171033 100001 170705 040704 104001 100002 000213 024733     00050 002022 005005 004374 100000 024711 027000 034401 014
00060  040401 024700 027000 040402 044031 170034 170705 100002     00070 026007 103001 143705 010444 103002 173705 024700 027
00100  033402 147033 010451 103002 173705 024700 027000 033402     00110 147033 010442 103002 173705 173034 070403 024722 027
00120  033402 147033 010431 103002 173705 173034 070403 024722     00130 027000 033402 147033 010420 033402 100035 147033 010
00140  100003 000205 011002 000100 010401 100004 000030 177725     00150 177733 177741 177751 177761 170034 040403 104004 000
00160  033402 143033 100031 024755 027000 125006
```

```
PAGE 21   $BOOKS1.M096B01.SERVESRC [1] SERVE: execute Procedure                    2/23/82   13:

  684.    000000 0  0  ! this is the main execution loop of the server process.  The server waits
  685.    000000 0  0  ! for incoming requests or system messages.
  686.    000000 0  0
  687.    000000 0  0  PROC execute;
  688.    000000 1  0    BEGIN
  689.    000000 1  1
  690.    000000 1  1      INT .pid[0:3],       ! requestor <process id>.
  691.    000000 1  1          system^message,
  692.    000000 1  1          status;
  693.    000000 1  1
  694.    000000 1  1      WHILE 1 DO ! loop on requests.
  695.    000003 1  1        BEGIN
  696.    000003 1  2
  697.    000003 1  2          ! read $RECEIVE file.
  698.    000003 1  2          CALL READUPDATE ( recv^fnum, recv^buf, message^len );
  699.    000014 1  2          CALL AWAITIO ( recv^fnum,, recv^cnt );
  700.    000024 1  2          IF >= THEN ! read a message.
  701.    000025 1  2            BEGIN
  702.    000025 1  3              system^message := >; ! save system message condition.
  703.    000032 1  3              CALL LASTRECEIVE ( pid );
  704.    000037 1  3              IF system^message THEN
  705.    000041 1  3                CALL analyze^system^message ( recv^buf, recv^cnt, pid )
  706.    000046 1  3              ELSE
  707.    000047 1  3                CALL process^user^request ( recv^buf, recv^cnt, pid );
  708.    000054 1  3            END;     ! read a message.
  709.    000054 1  2        END;     ! loop on requests.
  710.    000055 1  1    END; ! execute.

PID                              Variable        INT        L+001    Indirect
STATUS                           Variable        INT        L+003    Direct
SYSTEM^MESSAGE                   Variable        INT        L+002    Direct

00000  070404 024700 002006 040010 170026 005001 004004 024722     00010 002003 100034 024700 027000 070010 100000 070027 024
00020  002003 100024 024700 027000 014027 016002 100777 010401     00030 100000 044402 170401 100000 100002 024722 027000 040
00040  014406 170026 040027 170401 024722 027000 010405 170026     00050 040027 170401 024722 027000 010726 125003
```

```
  712.    000000 0  0  ! this is the "main".  This is where the primary/backup determination is
  713.    000000 0  0  ! made.
  714.    000000 0  0
  715.    000000 0  0  PROC serve MAIN;
  716.    000000 1  0
  717.    000000 1  0    BEGIN
  718.    000000 1  1
  719.    000000 1  1      INT  base = 'L' + 1,
  720.    000000 1  1           .ppdentry[0:8];
  721.    000000 1  1
  722.    000000 1  1      ! save stack^base for checkpointing.
  723.    000000 1  1      @stack^base := @base;
  724.    000005 1  1
  725.    000005 1  1      CALL ARMTRAP ( 0, -1 );
  726.    000011 1  1
  727.    000011 1  1      ! get process name.
  728.    000011 1  1      CALL GETCRTPID ( MYPID, ppdentry );
  729.    000015 1  1      CALL LOOKUPPROCESSNAME ( ppdentry );
  730.    000020 1  1      IF < THEN CALL ABEND;   ! not named.
  731.    000024 1  1
  732.    000024 1  1      ! calculate backup cpu number.  cpu's are paired 0-1, 2-3, 4-5, ...
  733.    000024 1  1      backup^cpu := MYPID.<0:7>;
  734.    000027 1  1      backup^cpu.<15> := NOT backup^cpu.<15>;
  735.    000040 1  1
  736.    000040 1  1      ! monitor all cpus.
  737.    000040 1  1      CALL MONITORCPUS ( -1 );
  738.    000043 1  1
  739.    000043 1  1      IF NOT ppdentry[4] THEN ! im the primary.
  740.    000046 1  1        BEGIN
  741.    000046 1  2          CALL read^start^up^message;
  742.    000047 1  2          CALL open^primarys^files;
  743.    000050 1  2          CALL create^backup ( backup^cpu );
  744.    000053 1  2          CALL execute;
  745.    000054 1  2        END
  746.    000054 1  1      ELSE                    ! im the backup.
  747.    000055 1  1        BEGIN
  748.    000055 1  2          ! wait for failure
  749.    000055 1  2          CALL CHECKMONITOR;
  750.    000057 1  2          CALL ABEND;
  751.    000062 1  2        END;
  752.    000062 1  1    END; ! serve.

BASE                          Variable        INT        L+001    Direct
PPDENTRY                      Variable        INT        L+001    Indirect


00000  070402 024700 002011 070401 044007 100000 100777 024711     00010 027000 027000 170401 024711 027000 170401 024700 027
00020  013003 000002 024711 027000 027000 030110 044000 040000     00030 006001 015402 100777 010401 100000 100001 070000 000
00040  100777 024700 027000 103004 143401 015407 027000 027000     00050 040000 024700 027000 027000 010405 027000 000107 000
00060  024711 027000 000002 024711 127000
```

ABEND	Proc			
ADDPID	Proc	INT		
ANALYZE^CHECKPOINT^STATUS	Proc			
ANALYZE^SYSTEM^MESSAGE	Proc			
ARMTRAP	Proc			
AWAITIO	Proc			
BACKUP^CPU	Variable	INT	G+000	Direct
BACKUP^PID	Variable	INT	G+001	Direct
BACKUP^UP	Variable	INT	G+006	Direct
BUF	Variable	INT	G+030	Indirect
CHECKMONITOR	Proc	INT		
CHECKOPEN	Proc			
CHECKPOINT	Proc	INT		
CREATE^BACKUP	Proc			
DB^FLAGS	Literal			%000040
DB^FNAME	Variable	INT	G+012	Direct
DB^FNUM	Variable	INT	G+011	Direct
DB^REC^KEY^LEN	Literal			%000030
DB^REC^KEY^OFF	Literal			%000000
DB^REC^LEN	Literal			%000400
DB^SYNC^DEPTH	Literal			%000001
DEBUG	Proc			
DELALLPIDS	Proc			
DELETE^REQUEST	Proc	INT		
DELPID	Proc			
EXECUTE	Proc			
FILEINFO	Proc			
GETCRTPID	Proc			
INSERT^REQUEST	Proc	INT		
KEYPOSITION	Proc			
LASTRECEIVE	Proc			
LOOKUPPID	Proc	INT		
LOOKUPPROCESSNAME	Proc			
MAX^REQSTRS	Literal			%000020
MESSAGE^LEN	Literal			%000404
MESSAGE^SIZE	Literal			%000202
MOM	Proc			
MONITORCPUS	Proc			
MYPID	Proc	INT		
NEWPROCESS	Proc			
NEXT^RECORD^REQUEST	Proc	INT		
NO^WAIT	Literal			%000001
OLD^REQUESTOR	Variable	INT	G+031	Direct
OPEN	Proc			
OPEN^BACKUPS^FILES	Proc			
OPEN^MSGS	Literal			%040000
OPEN^PRIMARYS^FILES	Proc			
PIDS	Variable	INT	G+035	Indirect
PROCESS^USER^REQUEST	Proc			
PROGRAMFILENAME	Proc			
PROTECTED	Literal			%000040
QUERY^REQUEST	Proc	INT		
READ	Proc			
READUPDATE	Proc			
READ^START^UP^MESSAGE	Proc			
RECORD	Literal			%000002

RECV^BUF	Variable	INT	G+026	Indirect
RECV^CNT	Variable	INT	G+027	Direct
RECV^FLAGS	Literal			%040001
RECV^FNUM	Variable	INT	G+010	Direct
RECV^SYNC^DEPTH	Literal			%000001
REPLY	Proc			
REPLY^BUF	Variable	INT	G+034	Indirect
REPLY^ERROR	Variable	INT	G+033	Indirect
REQUEST^TYPE	Literal			%000001
SERVE	Proc			
STACK^BASE	Variable	INT	G+007	Indirect
STOP	Proc			
STOP^COUNT	Variable	INT	G+005	Direct
SYNC	Literal			%000000
SYNC^COUNT	Variable	INT	G+032	Indirect
WRITE	Proc			
WRITEUPDATE	Proc			

PAGE 25 $BOOKS1.M096B01.SERVESRC [1] LOAD MAP 2/23/82 13:

PEP	BASE	LIMIT	ENTRY	ATTRIBUTES	NAME
002	000512	000600	000516		ADDPID
003	000370	000453	000370		ANALYZE^CHECKPOINT^STATUS
004	000733	001147	000733		ANALYZE^SYSTEM^MESSAGE
005	000311	000367	000311		CREATE^BACKUP
006	000662	000732	000662		DELALLPIDS
007	001227	001260	001227		DELETE^REQUEST
010	000601	000661	000601		DELPID
011	001511	001566	001511		EXECUTE
012	001205	001226	001205		INSERT^REQUEST
013	000454	000511	000454		LOOKUPPID
014	001261	001322	001261		NEXT^RECORD^REQUEST
015	000223	000310	000223		OPEN^BACKUPS^FILES
016	000205	000222	000205		OPEN^PRIMARYS^FILES
017	001323	001510	001323		PROCESS^USER^REQUEST
020	001150	001204	001150		QUERY^REQUEST
021	000023	000204	000023		READ^START^UP^MESSAGE
022	001567	001653	001567	M	SERVE

PEP	BASE	LIMIT	ENTRY	ATTRIBUTES	NAME
021	000023	000204	000023		READ^START^UP^MESSAGE
016	000205	000222	000205		OPEN^PRIMARYS^FILES
015	000223	000310	000223		OPEN^BACKUPS^FILES
005	000311	000367	000311		CREATE^BACKUP
003	000370	000453	000370		ANALYZE^CHECKPOINT^STATUS
013	000454	000511	000454		LOOKUPPID
002	000512	000600	000516		ADDPID
010	000601	000661	000601		DELPID
006	000662	000732	000662		DELALLPIDS
004	000733	001147	000733		ANALYZE^SYSTEM^MESSAGE
020	001150	001204	001150		QUERY^REQUEST
012	001205	001226	001205		INSERT^REQUEST
007	001227	001260	001227		DELETE^REQUEST
014	001261	001322	001261		NEXT^RECORD^REQUEST
017	001323	001510	001323		PROCESS^USER^REQUEST
011	001511	001566	001511		EXECUTE
022	001567	001653	001567	M	SERVE

```
Object file name is $BOOKS1.M096B01.serveobj
This object file will run on either a TNS or a TNS/II
Number of errors = 0
Number of warnings = 0
Primary global storage=30
Secondary global storage=503
Code size=921
Data area size=2  pages
Code area size=1  pages
Maximum symbol table space available = 24892, used = 1298
Maximum extended symbol table space available =     0, used = 0
Number of source lines=2142
Elapsed time -  00:00:39
```

APPENDIX C

SOURCE FOR $SYSTEM.SYSTEM.GPLDEFS

```
!?PAGE "T9600B00 - SIO PROCEDURES - DEFINITIONS"
!
! COPYRIGHT (C) 1985 TANDEM COMPUTERS INCORPORATED
! Protected as an unpublished work.
!
?SECTION  FCB^DEFS
!
! FCB SIZE IN WORDS.
!
LITERAL
     FCBSIZE = 60;
!
! DECLARE RUCB , PUCB, AND COMMON FCB.
!
DEFINE
     ALLOCATE^CBS ( RUCB^NAME , COMMON^FCB^NAME , NUM^FILES ) =
        INT .RUCB^NAME [ 0:65 ] :=
               ! RUCB PART.
               [ 62 , 1 , 27 * [ 0 ] , 62 , 32 * [ 0 ] ,
               ! PUCB PART.
                 4 , NUM^FILES , 4 + FCBSIZE ];
        INT .COMMON^FCB^NAME [ 0:FCBSIZE - 1 ] :=
               [ FCBSIZE * [ 0 ] ]#;
!
! DECLARE FCB.
!
DEFINE
     ALLOCATE^FCB ( FCB^NAME , PHYS^FILENAME ) =
        INT .FCB^NAME [ 0:FCBSIZE - 1 ] :=
         [ FCBSIZE , %000061 , -1 , %100000 , 0 , PHYS^FILENAME ,
                ( FCBSIZE - 17 ) * [ 0 ] ]#;
?SECTION  OPEN^DEFS
!
! OPEN ACCESS.
```

```
!
LITERAL
     READWRITE^ACCESS = 0,
     READ^ACCESS      = 1,
     WRITE^ACCESS     = 2;
!
! OPEN EXCLUSION.
!
LITERAL
     SHARE     = 0,
     EXCLUSIVE = 1,
     PROTECTED = 3;
!
! OPEN^FILE FLAGS                    V         111111
!                                    0123456789012345
                         !      1111111111222222222233
DEFINE                   !4567890123456789012345678901
     ABORT^OPENERR = %B00000000000000000000000000001D#,
     ABORT^XFERERR = %B00000000000000000000000000010D#,
     PRINT^ERR^MSG = %B00000000000000000000000000100D#,
     AUTO^CREATE   = %B00000000000000000000000001000D#,
     MUSTBENEW     = %B00000000000000000000000010000D#,
     PURGE^DATA    = %B00000000000000000000000100000D#,
     AUTO^TOF      = %B00000000000000000000001000000D#,
     NOWAIT        = %B00000000000000000000010000000D#,
     BLOCKED       = %B00000000000000000000100000000D#,
     VAR^FORMAT    = %B00000000000000000001000000000D#,
     READ^TRIM     = %B00000000000000000010000000000D#,
     WRITE^TRIM    = %B00000000000000000100000000000D#,
     WRITE^FOLD    = %B00000000000000001000000000000D#,
     WRITE^PAD     = %B00000000000000010000000000000D#,
     CRLF^BREAK    = %B00000000000000100000000000000D#;
?SECTION  SET^FILE^DEFS
!
! SET^FILE OPERATIONS.
!
LITERAL
      INIT^FILEFCB                =   0,
      !
      ASSIGN^FILENAME             =   1,
      ASSIGN^LOGICALFILENAME      =   2,
      ASSIGN^OPENACCESS           =   3,
      ASSIGN^OPENEXCLUSION        =   4,
      ASSIGN^RECORDLENGTH         =   5,
      ASSIGN^RECORDLEN            =   ASSIGN^RECORDLENGTH,
      ASSIGN^FILECODE             =   6,
      ASSIGN^PRIMARYEXTENTSIZE    =   7,
      ASSIGN^PRIEXT               =   ASSIGN^PRIMARYEXTENTSIZE,
      ASSIGN^SECONDARYEXTENTSIZE  =   8,
      ASSIGN^SECEXT               =   ASSIGN^SECONDARYEXTENTSIZE,
      ASSIGN^BLOCKLENGTH          =   9,
      ASSIGN^BLOCKBUFLEN          =   ASSIGN^BLOCKLENGTH,
```

```
!
SET^DUPFILE                    =  10,
SET^SYSTEMMESSAGES             =  11,
SET^OPENERSPID                 =  12,
SET^RCVUSEROPENREPLY           =  13,
SET^RCVOPENCNT                 =  14,
SET^RCVEOF                     =  15,
SET^USERFLAG                   =  16,
SET^ABORT^XFERERR              =  17,
SET^PRINT^ERR^MSG              =  18,
SET^READ^TRIM                  =  19,
SET^WRITE^TRIM                 =  20,
SET^WRITE^FOLD                 =  21,
SET^WRITE^PAD                  =  22,
SET^CRLF^BREAK                 =  23,
SET^PROMPT                     =  24,
SET^ERRORFILE                  =  25,
SET^PHYSIOOUT                  =  26,
SET^LOGIOOUT                   =  27,
SET^COUNTXFERRED               =  28,
SET^ERROR                      =  29,
SET^BREAKHIT                   =  30,
SET^TRACEBACK                  =  31,
!
SET^EDITREAD^REPOSITION        =  32,
!
FILE^FILENAME^ADDR             =  33,
FILE^LOGICALFILENAME^ADDR      =  34,
FILE^FNUM^ADDR                 =  35,
FILE^ERROR^ADDR                =  36,
FILE^USERFLAG^ADDR             =  37,
FILE^SEQNUM^ADDR               =  38,
FILE^FILEINFO                  =  39,
FILE^CREATED                   =  40,
FILE^FNUM                      =  41,
FILE^SEQNUM                    =  42,
FILE^ASSIGNMASK1               =  43,
FILE^ASSIGNMASK2               =  44,
FILE^FWDLINKFCB                =  45,
FILE^BWDLINKFCB                =  46,
!
SET^CHECKSUM                   =  47,
!
FILE^OPENERSPID^ADDR           =  48,
!
SET^SYSTEMMESSAGESMANY         =  49,
!
FILE^FCB^ADDR                  =  50,
!
MAX^OPERATION                  =  50,
!
```

```
      FILE^FILENAME             = ASSIGN^FILENAME             + 256,
      FILE^LOGICALFILENAME      = ASSIGN^LOGICALFILENAME      + 256,
      FILE^OPENACCESS           = ASSIGN^OPENACCESS           + 256,
      FILE^OPENEXCLUSION        = ASSIGN^OPENEXCLUSION        + 256,
      FILE^RECORDLEN            = ASSIGN^RECORDLENGTH         + 256,
      FILE^FILECODE             = ASSIGN^FILECODE             + 256,
      FILE^PRIEXT               = ASSIGN^PRIMARYEXTENTSIZE    + 256,
      FILE^SECEXT               = ASSIGN^SECONDARYEXTENTSIZE  + 256,
      FILE^BLOCKBUFLEN          = ASSIGN^BLOCKLENGTH          + 256,
      FILE^DUPFILE              = SET^DUPFILE                 + 256,
      FILE^SYSTEMMESSAGES       = SET^SYSTEMMESSAGES          + 256,
      FILE^OPENERSPID           = SET^OPENERSPID              + 256,
      FILE^RCVUSEROPENREPLY     = SET^RCVUSEROPENREPLY        + 256,
      FILE^RCVOPENCNT           = SET^RCVOPENCNT              + 256,
      FILE^RCVEOF               = SET^RCVEOF                  + 256,
      FILE^USERFLAG             = SET^USERFLAG                + 256,
      FILE^ABORT^XFERERR        = SET^ABORT^XFERERR           + 256,
      FILE^PRINT^ERR^MSG        = SET^PRINT^ERR^MSG           + 256,
      FILE^READ^TRIM            = SET^READ^TRIM               + 256,
      FILE^WRITE^TRIM           = SET^WRITE^TRIM              + 256,
      FILE^WRITE^FOLD           = SET^WRITE^FOLD              + 256,
      FILE^WRITE^PAD            = SET^WRITE^PAD               + 256,
      FILE^CRLF^BREAK           = SET^CRLF^BREAK              + 256,
      FILE^PROMPT               = SET^PROMPT                  + 256,
      FILE^ERRORFILE            = SET^ERRORFILE               + 256,
      FILE^PHYSIOOUT            = SET^PHYSIOOUT               + 256,
      FILE^LOGIOOUT             = SET^LOGIOOUT                + 256,
      FILE^COUNTXFERRED         = SET^COUNTXFERRED            + 256,
      FILE^ERROR                = SET^ERROR                   + 256,
      FILE^BREAKHIT             = SET^BREAKHIT                + 256,
      FILE^TRACEBACK            = SET^TRACEBACK               + 256,
      FILE^CHECKSUM             = SET^CHECKSUM                + 256,
      FILE^SYSTEMMESSAGESMANY   = SET^SYSTEMMESSAGESMANY      + 256;
?SECTION  ERROR^DEFS
!
! sio procedure errors.
!
LITERAL
     SIOERR^INVALIDPARAM     = 512, ! parameter is invalid.
     SIOERR^MISSINGFILENAME  = 513, ! filename not supplied.
     SIOERR^DEVNOTSUPPORTED  = 514, ! device not supported.
     SIOERR^INVALIDACCESS    = 515, ! access mode incompatible
                                    ! with device.
     SIOERR^INVALIDBUFADDR   = 516, ! buffer address not in lower
                                    ! 32k.
     SIOERR^INVALIDFILECODE  = 517, ! file code of file does not
                                    ! match assigned file code.
     SIOERR^BUFTOOSMALL      = 518, ! buffer to small for edit
                                    ! write (ie., less than 1024
                                    ! bytes) or buffer not
                                    ! sufficient for record
                                    ! length.
```

```
SIOERR^INVALIDBLKLENGTH = 519, ! assign block length > block
                               ! buffer length.
SIOERR^INVALIDRECLENGTH = 520, ! record length = 0 or record
                               ! length > maxrecordlength of
                               ! OPEN^FILE or record length
                               ! for $RECEIVE file < 14 or
                               ! record length > 254 and
                                 variable records specified
SIOERR^INVALIDEDITFILE  = 521, ! edit file is invalid.
SIOERR^FILEALREADYOPEN  = 522, ! OPEN^FILE called for file
already open.
SIOERR^EDITREADERR      = 523, ! edit read error.
SIOERR^FILENOTOPEN      = 524, ! file not open.
SIOERR^ACCESSVIOLATION  = 525, ! access not in effect for
                               ! requested operation.
SIOERR^NOSTACKSPACE     = 526, ! insufficient stack space
                               ! for temporary buffer
                                 allocation.
SIOERR^BLOCKINGREQD     = 527, ! block buffer required for
                                 nowait fold or pad.
SIOERR^EDITDIROVERFLOW  = 528, ! edit write directory
                               ! overflow.
SIOERR^INVALIDEDITWRITE = 529, ! write attempted after
                               ! directory has been written.
SIOERR^INVALIDRECVWRITE = 530, ! write to $RECEIVE does not
                               ! follow read.
SIOERR^CANTOPENRECV     = 531, ! can't open $RECEIVE for
                               ! break monitoring.
SIOERR^IORESTARTED      = 532, ! nowait io restarted.
SIOERR^INTERNAL         = 533, ! internal screwup.
SIOERR^CHECKSUMCOMM     = 534, ! common FCB checksum error.
SIOERR^CHECKSUM         = 535; ! file FCB checksum error.
```

APPENDIX D

SEQUENTIAL I/O FILE CONTROL BLOCK FORMAT

The following is the internal structure of the File Control Block (FCB) for the sequential I/O procedures described in Section 17.

NOTE

The FCB is included as a debugging aid only. Tandem Computers Incorporated reserves the right to make changes to the FCB structure. Therefore, this information must not be used to make program references to elements within the File Control Block.

```
!
! File Control Block (FCB) Structure Template.
!
STRUCT FCB^TMPL ( * );
  BEGIN
    INT SIZE,                    ! ( 0) size of FCB in words.
        NAMEOFFSET,              ! ( 1) word offset to name.
        FNUM;                    ! ( 2) GUARDIAN file number,
    !                                   -1 = closed.
    !
    ! create/open options group.
    !
    INT OPTIONS1,                ! ( 3) assign options.
        OPTIONS2,                ! ( 4) assign options.
        FILENAME [ 0:11 ],       ! ( 5) Tandem file name.
        !
        ! create options.
        !
        FCODE,                   ! (17) file code.
        PRIEXT,                  ! (18) primary extent size in
                                 ! pages.
```

```
    SECEXT,                     ! (19) secondary extent size in
    !                           !      pages.
    RECLEN,                     ! (20) logical record length.
    BLKBUFLEN,                  ! (21) block length from ASSIGN,
    !                           !      block buffer length
    !                           !      following OPEN^FILE.
    !
    ! open options.
    !
    OPENEXCLUSION,              ! (22) exclusion bits to OPEN.
    OPENACCESS;                 ! (23) access bits to OPEN.
!
! initializer group.
!
INT PUCB^POINTER,               ! (24) not used by SIO
                                ! procedures.
    SAMEFILELINK;               ! (25) not used by SIO
                                ! procedures.
!
! beginning of sio groups.
!
INT FWDLINK,                    ! (26) forward link.
    BWDLINK,                    ! (27) backward link.
    ADDR,                       ! (28) address of this FCB.
    COMMONFCBADDR,              ! (29) address of common FCB.
    ERROR;                      ! (30) last error.
!
! file FCB section.
!
INT DEVINFO,                    ! (31) file type, dev type, dev
                                !      subtype.
    OPENFLAGS1,                 ! (32) access mode, flags
                                !      parameters to OPEN^F&ILE.
    OPENFLAGS2,                 ! (33) flags parameters to
                                !      OPEN^FILE.
    XFERCNTL1,                  ! (34) iotype, sysbuflen,
                                !      interactive prompt.
    XFERCNTL2,                  ! (35) physioout, logioout,
                                !      write flush, retry count,
                                !      edit write control.
    DUPFCBADDR;                 ! (36) FCB address of file where
                                !      data read from this file
                                !      is to be written.
INT(32)
    LINENO;                     ! (37) line number from edit
                                !      read or ordinal record
                                !      count scaled by 1000.
    !
    ! Data Transfer/Blocking Group.
    !
INT BLKBUFADDR,                 ! (39) word address of block
                                !      buffer.
```

```
    BLKXFERCNT,                 ! (40) number of bytes to be
                                !      transferred between
                                !      device and target buffer.
    BLKREADCNT =                ! (40) number of bytes to be
      BLKXFERCNT,               !      read from device to
                                !      target buffer.
    BLKWRITECNT =               ! (40) number of bytes to be
      BLKXFERCNT,               !      written from target
                                !      buffer to device.
    BLKCNTXFERRED,              ! (41) number of bytes
                                !      transferred between
                                !      device and target buffer.
    BLKCNTREAD =                ! (41) number of bytes read into
      BLKCNTXFERRED,            !      target buffer.
    BLKCNTWRITTEN =             ! (41) number of bytes written
      BLKCNTXFERRED,            !      from target buffer.
    BLKNEXTREC,                 ! (42) (byte address) While
                                !      blocking/deblocking this
                                !      is the address of the
                                !      next record pointer in
                                !      the block buffer.
    USRBUFADDR,                 ! (43) byte address of user
                                !      buffer.
    USRWRCNT,                   ! (44) <write count> parameter
                                !      of WRITE^FILE, <prompt
                                !      count> parameter of
                                !      READ^FILE.
    USRRDCNT,                   !  (45) <max read count>
                                !       parameter of READ^FILE.
    TFOLDLEN,                   ! (46) terminal write fold
                                !      length (= physical record
                                !      length).
    USRCNTRD =                  ! (46) number of bytes read into
      TFOLDLEN,                 !      user buffer.
    PHYSXFERCNT,                ! (47) transfer count value
                                !      passed to file system in
                                !      SIO^PIO.
    PHYSIOCNTXFERRED,           ! (48) count transferred value
                                !      returned from file system
                                !      procedure.
    PHYSIOCNTRD =               ! (48) count read value returned
       PHYSIOCNTXFERRED,        !      from file system.
    PHYSIOCNTWR =               ! (48) count written value
       PHYSIOCNTXFERRED;        !      returned from file system
!
INT USERFLAG =                  ! (24) flag word to be set by
    PUCB^POINTER;               !      user.
!
! initializer group.
!
INT LOGICALFILENAME [ 0:3 ]; ! (49) logical file name of this
                             !      file to INITIALIZER.
```

```
! common FCB section.
!
    ! Break Group.
    !
INT BRKFCBADDR =              ! (31) FCB of file owning BREAK.
      DEVINFO,
    BRKMSG =                  ! (32:33) BREAK message buffer.
      OPENFLAGS1,
    BRKCNTL =                 ! (34) break control.
      XFERCNTL1,
    BRKLASTOWNER =            ! (35) BREAK last owner.
      XFERCNTL2;
    !
    ! $RECEIVE Group.
    !
    ! DUPFCBADDR skipped; was system messages mask.
    !
INT RCVCNTL =                 ! (37) $RECEIVE control.
      LINENO,
    PRIMARYPID [-1:-1] =      ! (38:41) Primary opener's
      LINENO,                 !         <process id>.
    BACKUPPID =               ! (42:45) Backup opener's
      BLKNEXTREC,             !         <process id>.
    REPLYCODE =               ! (46) $RECEIVE reply error code
      TFOLDLEN;
    ! Misc Group.
    !
INT COMMCNTL =                ! (47)
      PHYSXFERCNT,
    OPRQSTFCBADDR =           ! (48) FCB of file for which
      PHYSIOCNTXFERRED,       !      operator console messages
                              !      are displayed.
                              !      (see NO^ERROR, prompt).
    OPRQSTCOUNT =             ! (49) Count of number of
      LOGICALFILENAME,        !      operator messages
                              !      displayed.  (see
                              !      NO^ERROR, prompt).
    ERRFCBADDR [-1:-1] =      ! (50) FCB address of file where
      LOGICALFILENAME;        !      errors are to be reported
!
INT PXCNT,                    ! (53) Length of partial record
                              !      transferred between user
                              !      buffer and block buffer.
    PRCNT = PXCNT,            !      Partial read record
                              !      length.
    PWCNT = PXCNT;            !      Partial write record
                              !      length.
!
! The SYSMSGS[1234] words are used ONLY in the common FCB.
!
INT SYSMSGS1,                 ! (54) System messages to be
    SYSMSGS2,                 ! (55) passed back to caller.
```

```
        SYSMSGS3,                   ! (56)
        SYSMSGS4;                   ! (57)
    !
    ! INT SPARE1;                   ! (58) no-longer-unused FCB word
    INT EDIT^DIR^PAGE;              ! (58) If edit-file directory is
    INT E^DIR^STATE^UB =            !  over a page, these two words
                                    !  show where it is in the file.
          SYSMSGS4;                 ! (36) SIOWF.
    !
    INT SPARE1;                     ! (58) Unused FCB word.
    INT CHECKSUM;                   ! (59) Checksum.  If <> 0, check
    !
  END; ! FCB^TMPL.
!
! -- BIT FIELDS and ASSIGN BITS.
!
DEFINE
    !
    FILENAMESUPPLD  = <0>#,
      FCB^FILENAMESUPPLD        = FCB.OPTIONS1.FILENAMESUPPLD#,
    !
    PRIEXTSUPPLD    = <1>#,
      FCB^PRIEXTSUPPLD          = FCB.OPTIONS1.PRIEXTSUPPLD#,
    !
    SECEXTSUPPLD    = <2>#,
      FCB^SECEXTSUPPLD          = FCB.OPTIONS1.SECEXTSUPPLD#,
    !
    FCODESUPPLD     = <3>#,
      FCB^FCODESUPPLD           = FCB.OPTIONS1.FCODESUPPLD#,
    !
    EXCLUSIONSUPPLD = <4>#,
      FCB^EXCLUSIONSUPPLD       = FCB.OPTIONS1.EXCLUSIONSUPPLD#,
    !
    ACCESSSUPPLD    = <5>#,
      FCB^ACCESSSUPPLD          = FCB.OPTIONS1.ACCESSSUPPLD#,
    !
    RRECLENSUPPLD   = <6>#,
      FCB^RRECLENSUPPLD         = FCB.OPTIONS1.RRECLENSUPPLD#,
    !
    BLOCKLENSUPPLD  = <7>#,
      FCB^BLOCKLENSUPPLD        = FCB.OPTIONS1.BLOCKLENSUPPLD#;
!
! - OPEN EXCLUSION (FCB^OPENEXCLUSION)
!
DEFINE
     EXCLUSIONFIELD = <9:11>#,
       FCB^EXCLUSIONFIELD     = FCB.OPENEXCLUSION.EXCLUSIONFIELD#;
!
! - OPEN ACCESS (FCB^OPENACCESS)
!
DEFINE
     ACCESSFIELD    = <3:5>#,
```

```
       FCB^ACCESSFIELD          = FCB.OPENACCESS.ACCESSFIELD#;
!
! - DEVINFO.
!
DEFINE
     FILETYPE    =    <0:3>#,
       FCB^FILETYPE             = FCB.DEVINFO.FILETYPE#;
LITERAL
     UNSTR       = 0,
     ESEQ        = 1,
     REL         = 2,
     KSEQ        = 3,
     EDIT        = 4,
     ODDUNSTR    = 8;
DEFINE
     STRUCTFILE  = <2:3>#,       ! <>0 means structured file.
       FCB^STRUCTFILE           = FCB^DEVINFO.STRUCTFILE#;

DEFINE
     DEVTYPE     =    <4:9>#,
       FCB^DEVTYPE              = FCB.DEVINFO.DEVTYPE#;
LITERAL
     PROCESS     = 0,
     OPERATOR    = 1,
     RECEIVE     = 2,
     DISC        = 3,
     MAGTAPE     = 4,
     PRINTER     = 5,
     TERMINAL    = 6,
     DATACOMM    = 7,
     CARDRDR     = 8;
DEFINE
     DEVSUBTYPE = <10:15>#,
       FCB^DEVSUBTYPE           = FCB.DEVINFO.DEVSUBTYPE#;
!
! OPEN FLAGS. ( FCB.OPENFLAGS1 )
!
DEFINE
     FILECREATED       =  <0>#,  ! new file created.
       FCB^FILECREATED          = FCB.OPENFLAGS1.FILECREATED#;
DEFINE
     ACCESS         =    <1:3>#,  ! access mode.
       FCB^ACCESS               = FCB.OPENFLAGS1.ACCESS#;
LITERAL
       READACCESS       = 1,
       WRITEACCESS      = 2,
       READWRITEACCESS  = 3;
!
! allowable open flags 1 settings.
                           !          111111
LITERAL                    !0123456789012345
     ALLOWED^OPENFLAGS1 = %B0000111111111111;
```

```
!
! default open flags 1 settings.
                          !          111111
LITERAL                   !0123456789012345
     DEFAULT^OPENFLAGS1 = %B0000000000000000;
!
! OPEN FLAGS. ( FCB.OPENFLAGS2 )
!
DEFINE
     ABORTONOPENERROR = <15>#, ! abend on fatal error during open
       FCB^ABORTONOPENERROR   = FCB.OPENFLAGS2.ABORTONOPENERROR#;
DEFINE
     ABORTONXFERERROR = <14>#,  ! abend on fatal error during
                                ! data transfer.
       FCB^ABORTONXFERERROR   = FCB.OPENFLAGS2.ABORTONXFERERROR#;
DEFINE
     PRINTERRMSG      = <13>#,! print error message on fatal error
       FCB^PRINTERRMSG        = FCB.OPENFLAGS2.PRINTERRMSG#;
DEFINE
     AUTOCREATE       = <12>#,  ! create a file if write access.
       ! 0 = don't.
       ! 1 = do.
       FCB^AUTOCREATE         = FCB.OPENFLAGS2.AUTOCREATE#;
DEFINE
     FILEMUSTBENEW    = <11>#,  ! if autocreate = 1, no such file
                                ! may currently exist.
       ! 0 = old file is allowed.
       ! 1 = file must be new.
       FCB^FILEMUSTBENEW      = FCB.OPENFLAGS2.FILEMUSTBENEW#;
DEFINE
     WRITEPURGEDATA   = <10>#,  ! purge existing data.
       ! 0 = APPEND.
       ! 1 = PURGEDATA.
       FCB^WRITEPURGEDATA     = FCB.OPENFLAGS2.WRITEPURGEDATA#;
DEFINE
     AUTOTOF          =  <9>#,  ! auto page eject on open for
                                ! printer/process.
       ! 0 = NO
       ! 1 = YES
       FCB^AUTOTOF            = FCB.OPENFLAGS2.AUTOTOF#;
DEFINE
     NOWAITIO         =  <8>#,  ! open with nowait depth of 1.
       ! 0 = WAIT.
       ! 1 = NO-WAIT.
       FCB^NOWAITIO           = FCB.OPENFLAGS2.NOWAITIO#;
DEFINE
     BLOCKEDIO        =  <7>#,  ! blocked I/O.
       ! 0 = NOT BLOCKED
       ! 1 = BLOCKED
       FCB^BLOCKEDIO          = FCB.OPENFLAGS2.BLOCKEDIO#;
DEFINE
     VARFORMAT        =  <6>#,  ! variable length records.
```

```
        ! 0 = FIXED LENGTH
        ! 1 = VARIABLE LENGTH
        FCB^VARFORMAT           = FCB.OPENFLAGS2.VARFORMAT#;
DEFINE
     READTRIM          =  <5>#,  ! trim trailing blanks.
        ! 0 = NOTRIM
        ! 1 = TRIM
        FCB^READTRIM            = FCB.OPENFLAGS2.READTRIM#;
DEFINE
     WRITETRIM         =  <4>#,  ! trim trailing blanks.
        ! 0 = NOTRIM
        ! 1 = TRIM
        FCB^WRITETRIM           = FCB.OPENFLAGS2.WRITETRIM#;
DEFINE
     WRITEFOLD         =  <3>#,  ! fold write transfers greater
        ! 0 = TRUNCATE.             than write record length bytes
        ! 1 = FOLD.                 into multiple records.
        FCB^WRITEFOLD           = FCB.OPENFLAGS2.WRITEFOLD#;
DEFINE
     WRITEPAD          =  <2>#,  ! pad record with trailing blanks
        FCB^WRITEPAD            = FCB.OPENFLAGS2.WRITEPAD#;
DEFINE
     CRLFBREAK         =  <1>#,  ! CR/LF on break.
        ! 0 = NO CRLF ON BREAK.
        ! 1 = CRLF ON BREAK.
        FCB^CRLFBREAK           = FCB.OPENFLAGS2.CRLFBREAK#;
!
! allowable open flags 2 settings.
                              !          111111
LITERAL                       !0123456789012345
     ALLOWED^OPENFLAGS2 = %B1111111111111111;
!
! default open flags 2 settings.
                              !          111111
LITERAL                       !0123456789012345
     DEFAULT^OPENFLAGS2 = %B0101110001001111;
!
! TRANSFER CONTROL ( FCB.XFERCNTL1 )
!
DEFINE
     ERRORSET          = <0>#,
        ! ERROR SET INTO FCB VIA SET^FILE.
        FCB^ERRORSET            = FCB.XFERCNTL1.ERRORSET#;
DEFINE
     READIOTYPE        = <1:3>#,
        ! 0 = READ
        ! 1 = READUPDATE/REPLY
        ! 2 = EDITREAD
        ! 3 = WRITEREAD
        ! 7 = INVALID
        FCB^READIOTYPE          = FCB.XFERCNTL1.READIOTYPE#;
LITERAL
```

```
        STANDARDTYPE     = 0,
        RECEIVETYPE      = 1,
        EDITTYPE         = 2,
        INTERACTIVETYPE  = 3,
        INVALIDTYPE      = 7;
DEFINE
     WRITEIOTYPE      = <4:6>#,
        ! 0 = WRITE
        ! 1 = READUPDATE/REPLY
        ! 2 = EDITWRITE
        ! 7 = INVALID
        FCB^WRITEIOTYPE         = FCB.XFERCNTL1.WRITEIOTYPE#;
DEFINE
     SYSBUFLEN        = <7:8>#,  ! system buffer length / 1024.
        FCB^SYSBUFLEN            = FCB.XFERCNTL1.SYSBUFLEN#;
DEFINE
     PROMPT          = <9:15>#,  ! interactive prompt character.
        FCB^PROMPT               = FCB.XFERCNTL1.PROMPT#;
!
! TRANSFER CONTROL ( FCB.XFERCNTL2 )
!
DEFINE
     PHYSIOOUT          = <0>#,  ! physical (read-write) I/O
                                 ! outstanding.
        FCB^PHYSIOOUT            = FCB.XFERCNTL2.PHYSIOOUT#,
     READIOOUT          = <1>#,  ! logical read I/O outstanding.
        FCB^READIOOUT            = FCB.XFERCNTL2.READIOOUT#,
     WRITEIOOUT         = <2>#,  ! logical write I/O outstanding.
        FCB^WRITEIOOUT           = FCB.XFERCNTL2.WRITEIOOUT#,
     LOGIOOUT         = <1:2>#,  ! logical I/O outstanding.
        FCB^LOGIOOUT             = FCB.XFERCNTL2.LOGIOOUT#,
     WRITEFLUSH        =  <3>#,  ! block buffer flush operation
                                 ! in progress.
        FCB^WRITEFLUSH           = FCB.XFERCNTL2.WRITEFLUSH#,
     RETRYCOUNT       = <4:5>#,  ! I/O retry counter.
        FCB^RETRYCOUNT           = FCB.XFERCNTL2.RETRYCOUNT#,
     NOPARTIALREC     = <6>#,    ! blocks contain only full records.
        FCB^NOPARTIALREC         = FCB.XFERCNTL2.NOPARTIALREC#;
!
! TRANSFER CONTROL ( FCB.XFERCNTL2 )
!
! -- EDIT READ/WRITE CONTROL
!
DEFINE
     EDDIRWIP         =     <7>#, ! directory write in progress.
        FCB^EDDIRWIP             = FCB.XFERCNTL2.EDDIRWIP#,
     EDHALFSECTCNT    =  <8:11>#, ! number of half sectors written
                                  ! in current data page after next
                                  ! physical write.
        FCB^EDHALFSECTCNT        = FCB.XFERCNTL2.EDHALFSECTCNT#,
     EDDATABUFLEN     = <12:15>#, ! edit data buf size '>>'
                                  ! EDDBUFSHIFT (8).
```

```
      FCB^EDDATABUFLEN        = FCB.XFERCNTL2.EDDATABUFLEN#,
    EDREPOSITION   =     <7>#, ! user is repositioning edit file
                               ! (read op).
      FCB^EDREPOSITION        = FCB.XFERCNTL2.EDREPOSITION#;
!
! WRITE^FILE CONTROL OPERATION IN PROGRESS ( FCB.PHYSXFERCNT )
!
DEFINE
    CNTLINPROGRESS =     <0>#,
      FCB^CNTLINPROGRESS      = FCB.PHYSXFERCNT.CNTLINPROGRESS#,
    FORMSCNTLOP    =  <1:15>#,
      FCB^FORMSCNTLOP         = FCB.PHYSXFERCNT.FORMSCNTLOP#;
!
! BREAK CONTROL ( COMMFCB.BRKCNTL )
!
DEFINE
    BRKLASTMODE     = <0>#,    ! last break mode from SETMODE.
      COMMFCB^BRKLASTMODE     = COMMFCB.BRKCNTL.BRKLASTMODE#,
    BRKHIT          = <1>#,    ! BREAK key has been typed but
                               ! not tested.
      COMMFCB^BRKHIT          = COMMFCB.BRKCNTL.BRKHIT#,
    BRKFLUSH        = <2>#,    ! flush $RECEIVE BREAK message.
      COMMFCB^BRKFLUSH        = COMMFCB.BRKCNTL.BRKFLUSH#,
    BRKSTOLEN       = <3>#,    ! BREAK stolen away by another
                               ! process.
      COMMFCB^BRKSTOLEN       = COMMFCB.BRKCNTL.BRKSTOLEN#,
    BRKLDN       = <8:15>#, ! logical device number of terminal.
      COMMFCB^BRKLDN          = COMMFCB.BRKCNTL.BRKLDN#,
    COMMFCB^BRKARMED =         ! BREAK is armed.
      COMMFCB^BRKFCBADDR#;
!
! $RECEIVE CONTROL ( COMMFCB.RCVCNTL )
!
DEFINE
    RCVDATAOPEN      =   <0>#, ! $RECEIVE has been opened for
                               ! data transfer.
      COMMFCB^RCVDATAOPEN     = COMMFCB.RCVCNTL.RCVDATAOPEN#,
    RCVBRKOPEN       =   <1>#, ! $RECEIVE has been opened for
                               ! BREAK message reception.
      COMMFCB^RCVBRKOPEN      = COMMFCB.RCVCNTL.RCVBRKOPEN#,
    RCVOPENCNT       = <2:3>#, ! count of OPEN messages received
      COMMFCB^RCVOPENCNT      = COMMFCB.RCVCNTL.RCVOPENCNT#,
    RCVSTATE         =   <4>#,
      ! 0 = NEED READUPDATE.
      ! 1 = NEED REPLY.
      COMMFCB^RCVSTATE        = COMMFCB.RCVCNTL.RCVSTATE#,
    RCVUSEROPENREPLY =  <5>#,! user will reply to OPEN messages.
      ! 0 = SIO REPLIES.
      ! 1 = USER REPLIES.
     COMMFCB^RCVUSEROPENREPLY=COMMFCB.RCVCNTL.RCVUSEROPENREPLY#,
    RCVPSUEDOEOF     =   <6>#, ! pseudo-EOF. (N/A if user wants
                               ! CLOSE messages)
```

```
      ! 0 = EAT CLOSE MESSAGE.
      ! 1 = TURN LAST CLOSE MESSAGE INTO EOF.
      COMMFCB^RCVPSUEDOEOF    = COMMFCB.RCVCNTL.RCVPSUEDOEOF#,
    MONCPUMSG          = <2:3>#, ! user CPU Up/Down messages.
      COMMFCB^MONCPUMSG       = COMMFCB.SYSMSGS1.MONCPUMSG#,
    OPENMSG            =  <14>#, ! user wants OPEN messages.
      COMMFCB^OPENMSG         = COMMFCB.SYSMSGS2.OPENMSG#,
    CLOSEMSG           =  <15>#, ! user wants CLOSE messages.
      COMMFCB^CLOSEMSG        = COMMFCB.SYSMSGS2.CLOSEMSG#;
!
! COMMON CONTROL ( COMMFCB.COMMCNTL )
!
DEFINE
    CREATEINPROGRESS  = <0>#,  ! 1 during call to OPEN^FILE
                               ! while creating.
    COMMFCB^CREATEINPROGRESS=COMMFCB.COMMCNTL.CREATEINPROGRESS#,
    OPENINPROGRESS    = <1>#,  ! 1 during call to OPEN^FILE.
      COMMFCB^OPENINPROGRESS = COMMFCB.COMMCNTL.OPENINPROGRESS#,
    OPTYPE          = <0:1>#,  ! operation type.
      COMMFCB^OPTYPE          = COMMFCB.COMMCNTL.OPTYPE#,
    DEFAULTERRFILE    = <2>#,  ! defines default error reporting
                               ! file.
      ! 0 = home terminal.
      ! 1 = operator ($0).
      COMMFCB^DEFAULTERRFILE = COMMFCB.COMMCNTL.DEFAULTERRFILE#,
    TRACEBACK         = <3>#,  ! 1 = trace back to caller's P
                               ! when printing an error message.
      COMMFCB^TRACEBACK       = COMMFCB.COMMCNTL.TRACEBACK#;
```

INDEX

READER COMMENT CARD

Tandem welcomes your comments on the quality and usefulness of its software documentation. Does this manual serve your needs? If not, how could we improve it? Your comments will be forwarded to the writer for review and action, as appropriate.

If your answer to any of the questions below is "no," please supply detailed information, including page numbers, under Comments. Use additional sheets if necessary.

- ▶ Is this manual technically accurate? Yes ☐ No ☐
- ▶ Is information missing? Yes ☐ No ☐
- ▶ Are the organization and content clear? Yes ☐ No ☐
- ▶ Are the format and packaging convenient? Yes ☐ No ☐

Comments

Name Date

Company

Address

City/State Zip

GUARDIAN™ Operating System Programmer's Guide

NonStop™ Systems

82357 A00

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

B U S I N E S S R E P L Y M A I L

FIRST CLASS PERMIT NO. 482 CUPERTINO, CA, U.S.A.

POSTAGE WILL BE PAID BY ADDRESSEE

Tandem Computers Incorporated
Attn: Manager—Software Publications
Location 01, Department 6350
19333 Vallco Parkway
Cupertino CA 95014-9990

TAPE

TAPE

Tandem Computers Incorporated
19333 Vallco Parkway
Cupertino, CA 95014-2599