NonStop™ Systems

System Procedure Calls Reference Manual

Operating System Library

82359

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

System Procedure Calls Reference Manual

Abstract
This manual describes the syntax for all system procedure calls. This manual is for system and application programmers who need to call system procedures from their programs.

Product Version
GUARDIAN B00

Operating System Version
GUARDIAN B00 (NonStop Systems)

Part No. 82359 A00

March 1985

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

DOCUMENT HISTORY

Edition	Part Number	Operating System Version	Date
1st Edition	82359 A00	GUARDIAN B00	March 1985

New editions incorporate all updates issued since the previous edition. Update packages, which are issued between editions, contain additional and replacement pages that you should merge into the most recent edition of the manual.

The following are trademarks or service marks of Tandem Computers Incorporated:

AXCESS	BINDER	CROSSREF	DDL	DYNABUS
DYNAMITE	EDIT	ENABLE	ENCOMPASS	ENCORE
ENFORM	ENSCRIBE	ENTRY	ENTRY520	ENVOY
EXCHANGE	EXPAND	FOX	GUARDIAN	INSPECT
NonStop	NonStop 1+	NonStop II	NonStop TXP	PATHWAY
PCFORMAT	PERUSE	SNAX	Tandem	TAL
TGAL	THL	TIL	TMF	TRANSFER
T-TEXT	XRAY	XREF		

INFOSAT is a trademark in which both Tandem and American Satellite have rights.

HYPERchannel is a trademark of Network Systems Corporation.

IBM is a registered trademark of International Business Machines Corporation.

NEW AND CHANGED INFORMATION

This is a new publication for the NonStop systems. The existing two-volume GUARDIAN Operating System Programming Manual (Part Nos. 82336/82337) are being replaced by two separate manuals for the B00 release of the GUARDIAN operating system: the System Procedure Calls Reference Manual (Part No. 82358 A00) and the GUARDIAN Operating System Programmer's Guide (Part No. 82356 A00).

The GUARDIAN Operating System Programmer's Guide tells programmers how to use GUARDIAN calls (to those procedures previously described in the GUARDIAN Operating System Programming Manual) to accomplish various tasks.

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 (in order to aid the quick reference user), but the scope of the GUARDIAN Operating System Programmer's Guide has NOT been enlarged. The latter manual explains only how to perform tasks using those features now covered 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, continue to reside in the manuals for those products.

For the B00 release, the following new features were added to the procedure call information presented in this manual:

- 23 new procedures were added:

  ADDDSTTRANSITION
  CANCELPROCESSTIMEOUT
  COMPUTEJULIANDAYNO
  COMPUTETIMESTAMP
  CONVERTPROCESSTIME
  CONVERTTIMESTAMP
  CPUTIMES
  CURRENTSPACE
  DEBUGPROCESS
  DEVICEINFO2
  GETCPCBINFO
  INTERPRETJULIANDAYNO
  INTERPRETTIMESTAMP
  JULIANTIMESTAMP
  MYPROCESSTIME
  PROCESSFILESECURITY
  PROCESSORSTATUS
  PROCESSORTYPE
  PROCESSTIME
  REMOTETOSVERSION
  SETSYSTEMCLOCK
  SIGNALPROCESSTIMEOUT
  SYSTEMENTRYPOINTLABEL

- The following 16 procedures were modified:

  CANCELTIMEOUT
  CONTROL
  CREATE
  FILEINFO
  FILERECINFO
  GETSYSTEMNAME
  NEWPROCESS
  NEWPROCESSNOWAIT
  PRIORITY
  PROCESSINFO
  REFRESH
  SETLOOPTIMER
  SETMODE
  SETMODENOWAIT
  SIGNALTIMEOUT
  STOP

- Four new CONTROL operations was added:

  Operation = 1, 11, 12, 21

- Eighteen new functions for SETMODE and SETMODENOWAIT were added.

  <function> = 3, 5, 6, 22, 27-29, 37, 57, 67, 90-95, 111, 110, 113

- Three new subtypes were added:

  3 for the 3207 tape controller
  6 for the 5530 letter quality printer
  32 for the DTR printer

CONTENTS

FIGURES

TABLES

PREFACE

This reference manual describes the syntax of all system procedure calls for the NonStop systems.

This manual is for system and application programmers who need to call system procedures from their programs. Familiarity with the Transaction Application Language (TAL) or some other programming language is recommended. Before using this System Procedure Calls Reference Manual you should read:

- GUARDIAN Operating System Programmer's Guide (Part No. 82356 A00) for information about how to write programs using the GUARDIAN operating system procedure calls
- Transaction Application Language (TAL) Reference Manual (Part No. 82081 A00)

References to "DP1" and "DP2" in this manual have the following meanings:

- DP1 indicates the standard or base disc process for NonStop 1+ and NonStop systems.
- DP2 indicates an optional disc process for NonStop systems.

In this manual, Section 1 gives an overview of the categories of procedure calls and describes the format of a procedure call.

Section 2 describes each procedure call, except the sequential I/O (SIO) procedures, in alphabetic order.

Section 3 describes the SIO procedures in alphabetic order.

Appendix A describes the CONTROL operations you specify in order to perform device-dependent I/O operations.

Appendix B lists the device types and subtypes (such as discs, printers, terminals, and so forth) that are used on the Tandem system and are referenced by system procedure calls.

Appendix C lists the SETMODE functions you specify for use with I/O devices (such as setting disc file security, setting terminal interrupt characters, setting or clearing vertical tabs on a line printer, and so forth).

Appendix D contains a syntax summary of all system procedure calls.

Appendix E describes the ENFORM errors and their meanings.

Appendix F lists the numbered interprocess system messages sent to processes and their meanings.

Appendix G describes the SORT/MERGE errors and their meanings.

Appendix H lists the process names that are reserved for use by Tandem.

SYNTAX CONVENTIONS IN THIS MANUAL

The following is a summary of the conventions used in the 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 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 syntax items enclosed within the brackets or braces any number of times.
Punctuation	All punctuation marks and symbols not described above must be entered precisely as shown. If a punctuation mark or symbol appears enclosed in quotation marks, it is not a syntax descriptor; it is a required character, and you must enter it as shown.

NOTE

In procedure calls, input parameters (those that pass data from the calling program to the called procedure) are commented with an "i" (input) to the right of the parameter. Output parameters (those that return data from the called procedure to the calling program) are commented with an "o" (output) to the right of the parameter. When a parameter can be both input and output, it is commented with an "i" and an "o."

SECTION 1

INTRODUCTION TO SYSTEM PROCEDURE CALLS

System services are tasks that the GUARDIAN operating system or a subsystem performs on behalf of a program such as retrieving a record from a disc, writing a file to a tape, sending messages to other processes, or alerting your process to some kind of system malfunction.

Your programs can make use of these services by including calls to appropriate system procedures. For example, using the READ procedure allows a program to read data from a file.

To help you understand how to use the procedure call descriptions in this manual, Section 1 describes:

- The different types of system procedure calls
- A procedure call sample explaining the syntax
- The sequential I/O (SIO) procedures

This manual includes all the procedures that you can call from Transaction Application Language (TAL) programs and shows the syntax required to call these procedures from TAL. You can also call these procedures from FORTRAN, BASIC, or call some from COBOL programs. It is best, however, to be cautious when calling GUARDIAN operating system procedures from languages other than TAL, since some of the procedures can interfere with the underlying run-time environment already established by the language. When an operation (such as reading or writing) can be performed within the language itself, it is better to do so.

For information on translating the TAL calls in this manual to COBOL, refer to the COBOL Reference Manual. For information on translating the calls in this manual to FORTRAN, refer to the FORTRAN 77 Reference Manual. For information on translating the calls in this manual to BASIC, refer to the EXTENDED BASIC Reference Manual.

TYPES OF OPERATING SYSTEM PROCEDURE CALLS

Table 1-1 shows the types of system procedures that you can call in TAL programs and the manuals where you can find programming information about these different types of procedures.

Table 1-1. Procedure Call Types

Procedure Type	Action	Manuals
Checkpointing Facility	write information to a backup process.	*GUARDIAN Operating System Programmer's Guide*
ENFORM	communicate with the query processor.	*ENFORM User's Guide*
ENTRY/ ENTRY520	provides a method for creating and displaying application-defined forms on page-mode terminals.	*ENTRY Screen Formatter Operating and Programming Manual* and the *ENTRY520 Screen Formatter Operating and Programming Manual*
File System	perform operations, such as input and output, on files (this set of procedures includes ENSCRIBE procedures).	*GUARDIAN Operating System Programmer's Guide and the ENSCRIBE Programming Manual*
Formatter	format output data and convert input data.	*GUARDIAN Operating System Programmer's Guide*
Memory Management (Advanced Memory Management)	allocate and links to extended memory segments and pools; provides exclusive access to data.	*GUARDIAN Operating System Programmer's Guide*
Process Control	run, suspend, and stop programs.	*GUARDIAN Operating System Programmer's Guide*
Security	control access to processes and disc files.	*GUARDIAN Operating System Programmer's Guide*
Sequential I/O (SIO)	perform sequential input and output operations to files.	*GUARDIAN Operating System Programmer's Guide*
SORT/MERGE	control and execute sorting operations.	*SORT/MERGE User's Guide*
Spooler	control attributes and contents of jobs transmitted to printing devices.	*Spooler Programmer's Guide*
Transaction Monitoring Facility (TMF)	define and control TMF transactions.	*Transaction Monitoring Facility (TMF) Reference Manual*
Traps and Trap Handling	detect critical error conditions.	*GUARDIAN Operating System Programmer's Guide*
Utility	perform miscellaneous operations such as translating a number from displayed (string) form to integer form and vice versa and getting a timestamp.	*GUARDIAN Operating System Programmer's Guide*

The GUARDIAN Operating System Programmer's Guide describes how to use many of these procedures according to their function and type (the checkpointing facility, file system, formatter, memory management, process control, security, SIO, TMF, traps and trap handling, and uitility procedures). However, in the System Procedure Calls Reference Manual, the procedure descriptions are in alphabetic order for easy reference and are not arranged by type.

This manual provides the following information for each procedure:

- Syntax
- Parameter decriptions
- Condition codes
- Considerations
- Examples
- Manual references

SIO Procedures

The SIO procedures are a standardized set of procedures that handle I/O operations for different file types. The SIO procedures are a good tool for ensuring consistency in programs that access files sequentially. If you need to write to EDIT-format files, the SIO procedures provide the only programmatic method to do so.

Generally, you should not use these procedures and other GUARDIAN operating system I/O procedures together on the same file. For this reason, SIO procedures are described separately in Section 3.

SYNTAX OF A SYSTEM PROCEDURE CALL

An example of the syntax used in this manual is shown in Figure 1-1.

```
①                ②          ③                                    ④
{<length> := } FNAMECOLLAPSE ( <internal-name>                   ! i
{CALL        }                  ,<external-name> )               ! o

<length>                     returned value

  INT

  returns the number of bytes in <external-name>.

⑤
<internal-name>              input

  ⑥
  INT:ref:12

  ⑦
  is the name to be converted.   ...
```

① "<length> :="

This indicates that the procedure is a function procedure; it returns a value of the indicated type (in this case INT) when referenced in an expression. You can specify the variable as ⟨retval⟩, ⟨status⟩, ⟨error-code⟩, or some other appropriate name in other function procedure calls.

"CALL"

This is a TAL CALL statement. Any procedure that does not return a value must be invoked through the TAL CALL statement. In addition, you can use a CALL statement to invoke a function procedure if you do not need the returned value. You cannot invoke procedures from FORTRAN by using "CALL".

② This is the name of the system procedure that is called. It must appear in the program exactly as shown.

③ You must enclose the list of parameters in parentheses. Use commas to separate parameters when there is more than one.

If you omit optional parameters, the placeholder comma "," must be present, unless you omit the parameters from the end of the list.

④ The exclamation point indicates that a comment follows. The comment is either an "i" or an "o" (or both), which indicates that the parameter is either an input (i) or an output (o) parameter (or both). For a detailed description of input and output parameters, refer to the "Syntax Conventions" description at the beginning of this manual.

⑤ This line indicates whether the parameter is an input or an output parameter (or both).

Figure 1-1. Sample Procedure Call

⑥ This line indicates the parameter type:

INT	integer (one word)
INT(32)	doubleword integer (two words)
STRING	character string (one byte or half a word)
FIXED	quadword integer (four words)

The parameter type is followed by a colon. Additional information after the colon includes:

value — means the actual value or contents of a parameter are passed.

ref:x — means this is a reference parameter; that is, the address of the parameter is passed. (The statements within the program body must access the actual parameter contents indirectly through the parameter location.) "x" indicates how many elements the parameter contains. In this example, "12" indicates that the ⟨internal-name⟩ parameter contains 12 elements.

ref:* — means this is a reference parameter; how many elements returned varies according to how many elements requested.

EXT — means the parameter is a reference parameter accessed by an extended pointer.

NOTE

If a parameter is defined as "STRING:ref", a word-addressed variable (that is, an integer) can be passed for that parameter; the TAL compiler produces instructions to convert the word address to a byte address. An invalid address results if the word address is greater than 32767.

⑦ This describes the information that is passed or returned in the parameter.

S5023-001

Figure 1-1. Sample Procedure Call (Continued)

SECTION 2

SYSTEM PROCEDURE CALLS

This section contains detailed reference information for all system procedure calls. The information includes:

- A description of the call
- The syntax form for each call
- Parameter meanings of each call
- Condition code explanations for each call when applicable
- Considerations when using the call (added information about the procedure)
- Any applicable system messages
- Examples

ABEND PROCEDURE

The ABEND procedure is used to delete the calling process and to notify the creator process (with an ABEND system message) that an abnormal condition led to the deletion.

When ABEND executes, all open files associated with the deleted process automatically close. If the process owns BREAK, the process gives up BREAK ownership.

The syntax for ABEND is:

```
CALL ABEND;
```

Condition Code Settings

The condition code has no meaning following a call to ABEND.

Message

- Process Abnormal Deletion

  The creator of the aborted process receives a process abnormal deletion (ABEND) system message (-6) indicating that the deletion occurred. (Refer to Appendix F for a list of all system messages sent to processes.)

Example

```
CALL ABEND;
```

Related Programming Manual

For programming information about the ABEND process control procedure, refer to the GUARDIAN Operating System Programmer's Guide.

ABORTTRANSACTION PROCEDURE

ABORTTRANSACTION aborts and backs out a transaction. When the process that issued BEGINTRANSACTION (or its backup) calls this procedure, Transaction Monitoring Facility (TMF) backs out the data base changes made for the process's current-transaction identifier.

The syntax for ABORTTRANSACTION is:

```
<status> := ABORTTRANSACTION;

<status>                        returned value

  INT

  returns zero, if the call succeeds, or a file system error
  number.  (Refer to the System Messages Manual for a list of all
  file system errors.)
```

Condition Code Settings

The condition code has no meaning following a call to ABORTTRANSACTION.

Considerations

- Transaction's State After the Call to ABORTTRANSACTION

  When ABORTTRANSACTION returns, the transaction has not been backed out, but the transaction's state has changed from active to aborting. Later, the backout process will back out the transaction by restoring its before-images to all disc files that it changed. When backout is complete, the process releases the locks held for that transaction identifier.

  If the transaction is restarted with a new transaction identifier, it is not able to access any records locked by the aborted transaction until the following occurs: (a) backout is completed for the aborted transaction and (b) the locks held by the aborted transaction are released.

- Obtaining Main Memory for LCB

  If the procedure fails to obtain a link control block (LCB), then ABORTTRANSACTION fails with file system error 30.

- Requesting Process and Current-Transaction Identifier

  If the requesting process has no current-transaction identifier, then ABORTTRANSACTION fails with file system error 75.

- Invalid or Obsolete Transaction Identifier

  If the process that issues BEGINTRANSACTION (or its backup) did not begin the transaction, or if the transaction identifier is no longer in the system, this call returns file system error 78.

- When the Transaction Is Aborted

  Any disc process that receives a subsequent I/O request for the aborting transaction identifier rejects the request, and the call returns file system error 97.

Example

```
STATUS := ABORTTRANSACTION;
```

Related Programming Manual

For information about the ABORTTRANSACTION procedure, refer to the Transaction Monitoring Facility (TMF) Reference Manual.

ACTIVATEPROCESS PROCEDURE

The ACTIVATEPROCESS procedure is used to return a process or process pair from the suspended state to the ready state. (A process is put in the suspended state if it is the object of a call to the SUSPENDPROCESS procedure, or if it is suspended as the result of a SUSPEND command issued from the command interpreter.)

The syntax for ACTIVATEPROCESS is:

```
CALL ACTIVATEPROCESS ( <process-id> );              ! i

<process-id>                  input

  INT:ref:4

  is an array containing the process ID (PID) of the process to
  be activated.  If <process-id>[0:2] references a process pair
  and <process-id>[3] is specified as -1, then both members of
  the process pair are activated.
```

Condition Code Settings

< (CCL) indicates that ACTIVATEPROCESS failed, or no process designated as <process-id> exists.

= (CCE) indicates that the process is activated.

> (CCG) does not return from ACTIVATEPROCESS.

Considerations

- Process Accessor ID

  The caller of ACTIVATEPROCESS must be the super ID, the group manager of the process accessor ID, or a process with the same process accessor ID as the process or process pair being activated. Refer to the GUARDIAN Operating System Programmer's Guide for information about process accessor ID.

Example

```
CALL ACTIVATEPROCESS ( PROG^ID );  ! activate process.
```

Related Programming Manual

For programming information about the ACTIVATEPROCESS process control procedure, refer to the GUARDIAN Operating System Programmer's Guide.

ACTIVATERECEIVETRANSID PROCEDURE

ACTIVATERECEIVETRANSID is used to code $RECEIVE-queuing servers--that is, servers that can read requests from $RECEIVE before replying to previously read $RECEIVE requests. When a server calls this procedure with a message tag obtained by a call to the LASTRECEIVE or RECEIVEINFO file system procedure, the transaction identifier of the message associated with the tag becomes the current-transaction identifier for the server process. This multithreaded function provides a server process with the ability to concurrently serve more than one requester.

The syntax for ACTIVATERECEIVETRANSID is:

```
CALL ACTIVATERECEIVETRANSID ( <message-tag> );        ! i

<message-tag>                 input

  INT:value

  identifies a message request from the group of requests that
  are currently queued by the server; it is the same parameter
  that is passed by the server to the REPLY procedure.  The
  message tag must be an integer between 0 and (receivedepth-1),
  inclusive, that is currently associated with a queued message.
```

Condition Code Settings

< (CCL) indicates an error.

= (CCE) indicates that ACTIVATERECEIVETRANSID was successful.

> (CCG) does not return from ACTIVATERECEIVETRANSID.

ACTIVATERECEIVETRANSID returns a condition code only.

Example

```
CALL ACTIVATERECEIVETRANSID ( MSG^ID );
```

Related Programming Manual

For programming information about the ACTIVATERECEIVETRANSID procedure, refer to the Transaction Monitoring Facility (TMF) Reference Manual.

ADDDSTTRANSITION PROCEDURE

The ADDDSTTRANSITION procedure allows a user with a super group ID to add an entry to the daylight savings time (DST) transition table.

NOTE

The DST transition table must be loaded in time sequence and with no gaps (see "Considerations").

The syntax for ADDDSTTRANSITION is:

```
CALL ADDDSTTRANSITION ( <low-gmt>                    ! i
                       ,<high-gmt>                   ! i
                       ,<offset>  );                 ! i
```

<low-gmt> input

FIXED

is the Greenwich mean time (GMT) when <offset> is first applicable (this form is the same as the form used for COMPUTETIMESTAMP). Except for the first call, the <low-gmt> of each call must be the same as the <high-gmt> of the previous call. This implies that many calls have an <offset> parameter of 0.

<high-gmt> input

FIXED:value

is the GMT when <offset> is no longer applicable.

<offset> input

INT:value

is a value of <offset> in seconds:

local civil time (LCT) := local standard time (LST) + <offset>

Condition Code Settings

< (CCL) indicates that you either:

- do not have a super ID, user identification
- loaded the DST table inconsistently (that is, the DST table contains gaps or an overlap of entries)
- were loading the DST table at the same time someone else was loading the DST table.

= (CCE) indicates that the DST table was loaded successfully.

> (CCG) does not return from ADDDSTTRANSITION.

Considerations

- Loading the DST Transition Table With No Time Gaps

  Except for the first call, the DST transition table must be loaded in time sequence with no gaps, for example, if you load the following:

  First Entry

  ```
  1980 April 1, 2:00,
  1980 Oct   1, 2:00, 1:00
  ```

  Second Entry

  ```
  1980 Oct   1, 2:00,
  1981 April 1, 2:00, 0:00
  ```

  Next Entry (must be)

  ```
  1981 April 1, 2:00,
           .
           .
  ```

Example

```
CALL ADDDSTTRANSITION ( LOW , HIGH , OFFSET );
```

Related Programming Manual

None

ALLOCATESEGMENT PROCEDURE

The ALLOCATESEGMENT procedure allocates an extended data segment for use by the calling process.

NOTE

The call to ALLOCATESEGMENT must be followed by a call to USESEGMENT in order to make the extended memory accessible to the program. (Although you can have multiple extended segments, you can only access them one at a time.)

The syntax for ALLOCATESEGMENT is:

```
<status> := ALLOCATESEGMENT ( <segment-id>                ! i
                            ,[ <segment-size> ]           ! i
                            ,[ <filename> ]               ! i
                            ,[ <pin> ] );                 ! i
```

<status> returned value

INT

returns a status word having one of the following values:

0	= no error
1-999	= file system error related to the CREATE or the OPEN of the swap file (see <filename> parameter)
-1	= illegal <segment-id>
-2	= illegal <segment-size>
-3	= bounds violation on <filename>
-4	= illegal combination of options
-5	= unable to allocate segment space
-6	= unable to allocate segment page table space
-7	= security violation on attempt to share segment
-8	= <pin> does not exist
-9	= <pin> does not have the segment allocated
-10	= trying to share segment with self

→

<segment-id> input

INT:value

is the number by which the process chooses to refer to the segment. Segment IDs are in the following ranges:

0-1023	can be specified by user processes.
Other IDs	are reserved for Tandem-supplied software.

No process can supply a segment ID greater than 2047.

<segment-size> input

INT(32):value

is the number of bytes that the segment must hold. This value must be greater than 0 and less than %777777777D. If you do not supply this parameter, then you must give the <pin> parameter.

<filename> input

INT:ref:12

if present, is the name of a "swap file" to be associated with the segment. If the file exists, all data in the file is used as initial data for the segment. If the file does not exist, one is created. If the process terminates without deallocating the segment, any data still in memory is written back out to the file. ALLOCATESEGMENT must be able to allocate a sufficient number of file extents to contain all memory in the segment.

The parameter can be a volume name with a blank subvolume and file; ALLOCATESEGMENT allocates a temporary swap file on the indicated volume.

If you do not specify the parameter, ALLOCATESEGMENT uses the volume of the data stack swap file to create a temporary swap file for the new segment.

<pin>	input
INT:value designates that the segment specified by <segment-id> is to be shared with the process specified by <pin>. In order for this to occur, one of the following must be true: 1) the processes must execute in the same processor and must share the same access ID, 2) this process's access ID must be the group manager for the other's access ID, or 3) this process's access ID must be the super ID.	

Condition Code Settings

The condition code has no meaning following a call to ALLOCATESEGMENT (see the <status> parameter definition).

Considerations

- Existing Temporary File Name

  The <filename> parameter can specify an existing temporary file name allowing the application program to control file attributes (such as extent sizes or the clear-on-purge attribute): the file is automatically purged when the segment is deallocated or the application terminated.

- Preventing Automatic Temporary File Purge

  ALLOCATESEGMENT opens the swap file in a READ/WRITE/protected manner. A process can prevent the automatic file purge of a temporary swap file by opening the file for READ-only/shared access before the segment is deallocated.

Example

```
STATUS := ALLOCATESEGMENT ( SEGMENT^ID , SEG^SIZE , SWAP^FILE );
                          !  standard call to create a user segment;
                          !  "swap^file" parameter can be omitted.
```

Related Programming Manual

For programming information about the ALLOCATESEGMENT memory management procedure, refer to the GUARDIAN Operating System Programmer's Guide.

ALTERPRIORITY PROCEDURE

The ALTERPRIORITY procedure is used to change the execution priority of a process or process pair.

A process or process pair has two priority values: the initial priority value and the current priority value. ALTERPRIORITY changes both priority values to the specified value.

The syntax for ALTERPRIORITY is:

```
CALL ALTERPRIORITY ( <process-id>                ! i
                   ,<priority> );                ! i
```

<process-id> input

INT:ref:4

is an array containing the process ID of the process whose execution priority is to be changed. If <process-id>[0:2] references a process pair and <process-id>[3] is specified as -1, then the call applies to both members of the process pair.

<priority> input

INT:value

is a new execution priority value in the range of {1:199} for <process-id>.

Condition Code Settings

<	(CCL)	indicates that ALTERPRIORITY failed, or no process designated as <process-id> exists.
=	(CCE)	indicates that the priority of the process is altered.
>	(CCG)	does not return from ALTERPRIORITY.

Considerations

- Process Accessor ID

  The caller of ALTERPRIORITY must be the super ID, be the group manager of the process accessor ID, or a process with the same process accessor ID as the process or process pair priority being changed. Refer to the GUARDIAN Operating System Programmer's Guide for further information about the process accessor ID.

Example

```
CALL ALTERPRIORITY ( PID , PRI );
```

Related Programming Manual

None

ARMTRAP PROCEDURE

The ARMTRAP procedure is used to specify a location within the application program where execution begins if a trap occurs. The program can use information passed to investigate the cause of the error.

The syntax for ARMTRAP is:

```
CALL ARMTRAP ( <traphandlr-addr>                    ! i
             ,<trapstack-addr> );                   ! i
```

<traphandlr-addr> input

INT:value

is a label (nonzero P register value) that identifies a statement in the program where control transfers if a trap occurs.

If 0 is specified for <traphandlr-addr>, this means to reset the trap mechanism after a trap occurs, thus causing the process to restart. The process's registers at the time of the restart are set to the values indicated by the following 'L' relative locations:

```
'L'[-6] = unused
'L'[-5] = new value for space ID in stack marker ENV
          format: LS = .<4>, CS = .<7>, index = .<11:15>
          (see Considerations)
'L'[-4] = trap number
'L'[-3] = new value for S register
'L'[-2] = new value for P register
'L'[-1] = new value of hardware ENV register
'L'[0]  = new value for L register
'L'[1]  = new value for R0
'L'[2]  = new value for R1
'L'[3]  = new value for R2
'L'[4]  = new value for R3
'L'[5]  = new value for R4
'L'[6]  = new value for R5
'L'[7]  = new value for R6
'L'[8]  = new value for R7.
```

NOTE

'L'[-5] and 'L'[-1] are combined into new hardware ENV.

→

<trapstack-addr> input

INT:value

is an address specifying the local data area for the application process's trap handler. <trapstack-addr> also indicates where the trap number and stack marker at the time of the trap are passed to the application process. After a trap occurs, 'S' and 'L' are set to <trapstack-addr> plus six; the six words starting at <trapstack-addr> plus one are (given relative to the new 'L' setting):

'L'[-6] is unused
'L'[-5] is the stack marker ENV register with a space ID at the time of the trap
'L'[-4] is the trap number: 0 = illegal address reference
1 = instruction failure
2 = arithmetic overflow
3 = stack overflow
4 = process loop timer timeout
11 = memory manager read error
12 = no memory available
13 = uncorrectable memory error

'L'[-3] is the value of 'S' at the time of the trap; it is %177777 if the trap occurs while executing in system code or system library
'L'[-2] is the value of 'P' at the time of the trap. The 'P' value associated with the space ID in 'L'[-5] completely identifies the location of the trap.
'L'[-1] is the value of the hardware 'ENV' at the time of the trap
'L' is the value of 'L' at the time of the trap.

If <trapstack-addr> is passed as a value < 0, then any trap results in the process being stopped with an abnormal deletion indication (that is, ABEND message).

Condition Code Settings

The condition code has no meaning following a call to ARMTRAP.

Considerations

- Space ID, Stack Marker ENV and Hardware ENV Register

  Space ID consists of a code space bit that signifies system or user space, a library space bit that signifies code or library space, and a 5-bit index to indicate which segment of the 32 possible you are referring. Refer to the System Description Manual for more information about space IDs.

  On a procedure call, the space ID of the calling procedure is placed into the stack marker ENV register 'L'[-1]. At the time of a trap, the stack marker ENV register at 'L'[-1] contains the space ID of the location of the trap. When exiting the trap handler, execution resumes at the location identified by the 'P' register at 'L'[-2] and space ID in the stack at 'L'[-5]. To change the location of where execution is to resume, you need to change the 'P' register value as well as the space ID in the stack at 'L'[-5] to reflect the new location within your program. Refer to the following Consideration, "Calling ARMTRAP When Using Multi-Segment Programs."

- Calling ARMTRAP When Using Multisegment Programs

  The call to ARMTRAP that initially arms a trap handler must be in the same segment as the trap handling procedure when using multi-segment programs. Existing trap handlers in programs that become multisegment programs might not have to be modified. The only time a trap handler needs to be modified is when you want to access the space ID or change the resume location of a multi-segment process after a trap.

- Exiting an Application Process's Trap Handler Procedure

  If the application process's trap handling procedure is entered because of a trap, an exit from the procedure must be through a call to ARMTRAP, with <traphandlr-addr> specified as "0." The procedure must use ARMTRAP (it cannot use EXIT) to exit through the stack marker at the current 'L' register location. (This would result in an invalid 'S' register setting following the exit.)

- Trap Handler and a Call to a System Procedure

  If the trap handler calls any operating system procedure, at least 350 words must be available between the trap address value, specified to ARMTRAP, and the last word in the application's data area or 'G'[32767], whichever is less.

- Trap Handler Data Area

  Since the area below the stack pointer can be used internally by the operating system before ARMTRAP is called, do not locate the trap handler data area below the memory stack pointer.

- Base Address Equivalencing and Declaring Local Variables

  Any local variables in the application program's trap handling procedure must be declared relative to the L register by using base-address equivalencing. Base-address equivalencing relative to the L register is of the form:

  ```
  <type> { [ . ] <name> = 'L' [ { + | - } <word-offset> ] } ...
  ```

  where

  <type> is the data type of the variable <name>.

  <word-offset> specifies a positive or negative offset from the L register where the variable exists.

  NOTE

  Variables declared in this form cannot be initialized.

  The trap handling procedure must contain a statement that explicitly allocates storage for any locally declared variables (see Consideration, "Saving the Register Stack Registers").

- Saving the Register Stack Registers

  The stack registers (that is, R0-R7) contains the values they had at the time of the trap upon entry to the application process's trap handler. To save these values, the first statement of the trap handler must be:

  ```
  CODE( PUSH %777 )
  ```

  This will save the register stack contents. Local storage can then be allocated by adding the appropriate value to 'S' through a statement of the form:

  ```
  CODE ( ADDS <num-locals> )
  ```

  <num-locals> is a LITERAL defining the number of words of local storage needed.

- Value for the P Register

  The value for the P register at the time of the trap depends upon the trap condition:

trap	P register
0	I
1	I
2	I + 1
3	?
4	I
11	I
12	I
13	?

  where

  I = the address of the instruction being executed at the time of the trap.
  ? = undefined.

- Overflow Trap and the Process Continuing

  If the trap handler is entered because of an overflow trap and the application process intends to continue processing, then the overflow bit in the ENV register value in 'L'[-1] of the trap handler must be set to zero before the trap mechanism is rearmed. Otherwise, another overflow trap immediately occurs.

- How to Avoid Writing Over the Application's Data Stack

  If 'L'[-3] (value of 'S' at time of trap) is %177777, the trap handler should not reset traps without first changing 'L'[-3] to a more appropriate value. Otherwise, G[0] through G[10] of the application's data stack is overwritten.

Example

```
PROC TRAPPROC:
  BEGIN
  CALL ARMTRAP ( @TRAP, $LMIN ( LASTADDR , %77777 ) - 500 );
    ! setting the trap.
  EXIT
  TRAP;
    .
    .
    .
  END

PROC MAIN PROC;
  BEGIN
    CALL TRAPPROC;
      .
      .
      .
  END;
```

In the above example, @TRAP is the label at the beginning of the Tandem Application Language trap handler procedure where control is transferred if a trap occurs. The $LMIN expression is the address of the local data area where the trap handler runs (its data area). Refer to the GUARDIAN Operating System Programmer's Guide for a detailed example using the ARMTRAP procedure.

Related Programming Manual

For programming information about the ARMTRAP trap handling procedure, refer to the GUARDIAN Operating System Programmer's Guide.

AWAITIO PROCEDURE

The AWAITIO procedure is used to complete a previously initiated I/O operation. Use AWAITIO to:

- Wait for the operation to complete on:

  a. A particular file

     Application process execution suspends until the completion occurs. A timeout is considered to be a completion in this case.

  b. Any file or for a timeout to occur.

     A timeout is not considered a completion in this case.

- Check for the operation to complete on:

  a. A particular file

     The call to AWAITIO immediately returns to the application process, regardless of whether there is a completion or not. (If there is no completion, an error indication returns.)

  b. Any file

You can specify a time limit if AWAITIO is used to wait for a completion to increase the time allotted to completing the waited-for operation.

The syntax for AWAITIO is:

```
CALL AWAITIO ( <filenum>                          ! i, o
             ,[ <buffer-addr> ]                   ! o
             ,[ <count-transferred> ]             ! o
             ,[ <tag> ]                           ! o
             ,[ <timelimit> ] );                  ! i
```

<filenum> input, output

INT:ref:1

is the number of an open file. If a particular <filenum> is passed, AWAITIO applies to that file.

→

If <filenum> is passed as -1, the call to AWAITIO applies to the oldest incomplete operation pending on each file. The specific action depends on the value of the <timelimit> parameter (see the <timelimit> parameter below).

AWAITIO returns into <filenum> the file number associated with the completed operation.

<buffer-addr> output

INT:ref:1

returns the address of the <buffer> specified when the operation was initiated.

NOTE

If the actual parameter is used as an address pointer to the returned data and is declared in the form INT .<buffer-addr>, then it should be returned to AWAITIO in the form @<buffer-addr>.

<count-transferred> output

INT:ref:1

returns the count of the number of bytes transferred because of the associated operation.

<tag> output

INT(32):ref:1

returns the application-defined tag that was stored by the system when the I/O operation associated with this completion was initiated.

→

<timelimit> input

INT(32):value

indicates whether the process waits for completion instead of checking for completion. If <timelimit> is passed as:

<>	0D	a wait-for-completion is specified. <timelimit> then specifies the maximum time (in .01-second units) that the application process can wait (that is, be suspended) for completion of a waited-for operation.
=	-1D	an indefinite wait is indicated.
=	0D	a check for completion is specified. AWAITIO immediately returns to the caller, regardless of whether or not an I/O completion occurs.
<	-1D	file system error 22 occurs.
omitted		no limit exists, and an indefinite wait is indicated.

Condition Code Settings

< (CCL) indicates that an error occurred (call FILEINFO).

= (CCE) indicates that an I/O operation completed.

> (CCG) indicates that a warning occurred (call FILEINFO).

Considerations

- Completing Nowait Calls

  Each nowait operation initiated must be completed with a corresponding call to AWAITIO.

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

  --If AWAITIO is used to check for completion (<timelimit> = 0D) or used to wait on any file (<filenum> = - 1), completing AWAITIO does not necessarily indicate a completion.

If you perform an operation using one of the following procedure calls with a file opened nowait, you must complete the operation with a call to the AWAITIO procedure:

CONTROL	SETMODENOWAIT
CONTROLBUF	
	UNLOCKFILE
LOCKFILE	UNLOCKREC
LOCKREC	
	WRITE
READ	WRITEREAD
READLOCK	WRITEUPDATE
READUPDATE	WRITEUPDATEUNLOCK
READUPDATELOCK	

- Order of I/O Completion With SETMODE 30

  Specifying SETMODE 30 allows nowait I/O operations to complete in any order. However, I/O operations that complete at the same time return in the order issued. An application process that uses this option can use the <tag> parameter to keep track of multiple I/O operations associated with a file OPEN.

- Order of I/O Completion Without SETMODE 30

  If SETMODE 30 is not set, the oldest incomplete I/O operation always completes first; therefore, AWAITIO completes I/O operations associated with the particular open of a file in the same order as initiated.

- Error Handling

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

- Operation Timed Out

  If an error indication returns and a subsequent call to FILEINFO returns error 40, the operation is considered incomplete and AWAITIO must be called again.

- WRITE Buffers

  The contents of a buffer between a nowait initiation (for example, a call to WRITE) and the corresponding nowait completion (that is, a call to AWAITIO) should not be altered.

- No Nowait Operation

  You should not call AWAITIO unless you initiate a nowait operation prior to the call; otherwise, an error indication returns (CCL). A subsequent call to FILEINFO returns error 26.

- AWAITIO Completion Summary

  How AWAITIO completes depends on whether the <filenum> parameter specifies a particular file or any file and on what the value of <timelimit> is when passed with the call. The action taken by AWAITIO for each combination of <filenum> and <timelimit> is summarized in Figure 2-1.

- AWAITIO Operation

  The operation of the AWAITIO procedure is shown in Figure 2-2.

	⟨time limit⟩ = 0	⟨time limit⟩ ≠ 0
Particular File **⟨fn⟩ = ⟨file num⟩**	**CHECK** for any ⟨file num⟩ I/O completion. **COMPLETION** File number is returned in ⟨fn⟩. Tag of completed call is returned in ⟨tag⟩. **NO COMPLETION** CCL (error 40) is returned. File number returned is in ⟨fn⟩. No I/O operation is canceled.	**WAIT** for any ⟨file num⟩ I/O completion. **COMPLETION** File number is returned in ⟨fn⟩. Tag of completed call is returned in ⟨tag⟩. **NO COMPLETION** CCL (error 40) is returned. File number is returned in ⟨fn⟩. ***Oldest*** ⟨file num⟩ I/O operation is canceled. Tag of canceled call is returned in ⟨tag⟩.
Any File **⟨fn⟩ = −1**	**CHECK** for any I/O completion on any open file. **COMPLETION** File number of completed call is returned in ⟨fn⟩. Tag of completed call is returned in ⟨tag⟩. **NO COMPLETION** CCL (error 40) is returned. The value −1 is returned in ⟨fn⟩. No I/O operation is canceled.	**WAIT** for any I/O completion on any open file. **COMPLETION** File number of completed call is returned in ⟨fn⟩. Tag of completed call is returned in ⟨tag⟩. **NO COMPLETION** CCL (error 40) is returned. The value −1 is returned in ⟨fn⟩. No I/O operation is canceled.

Notes: ⟨fn⟩ = ⟨file number⟩
SETMODE 30 Set

S5023-002

Figure 2-1. AWAITIO Action

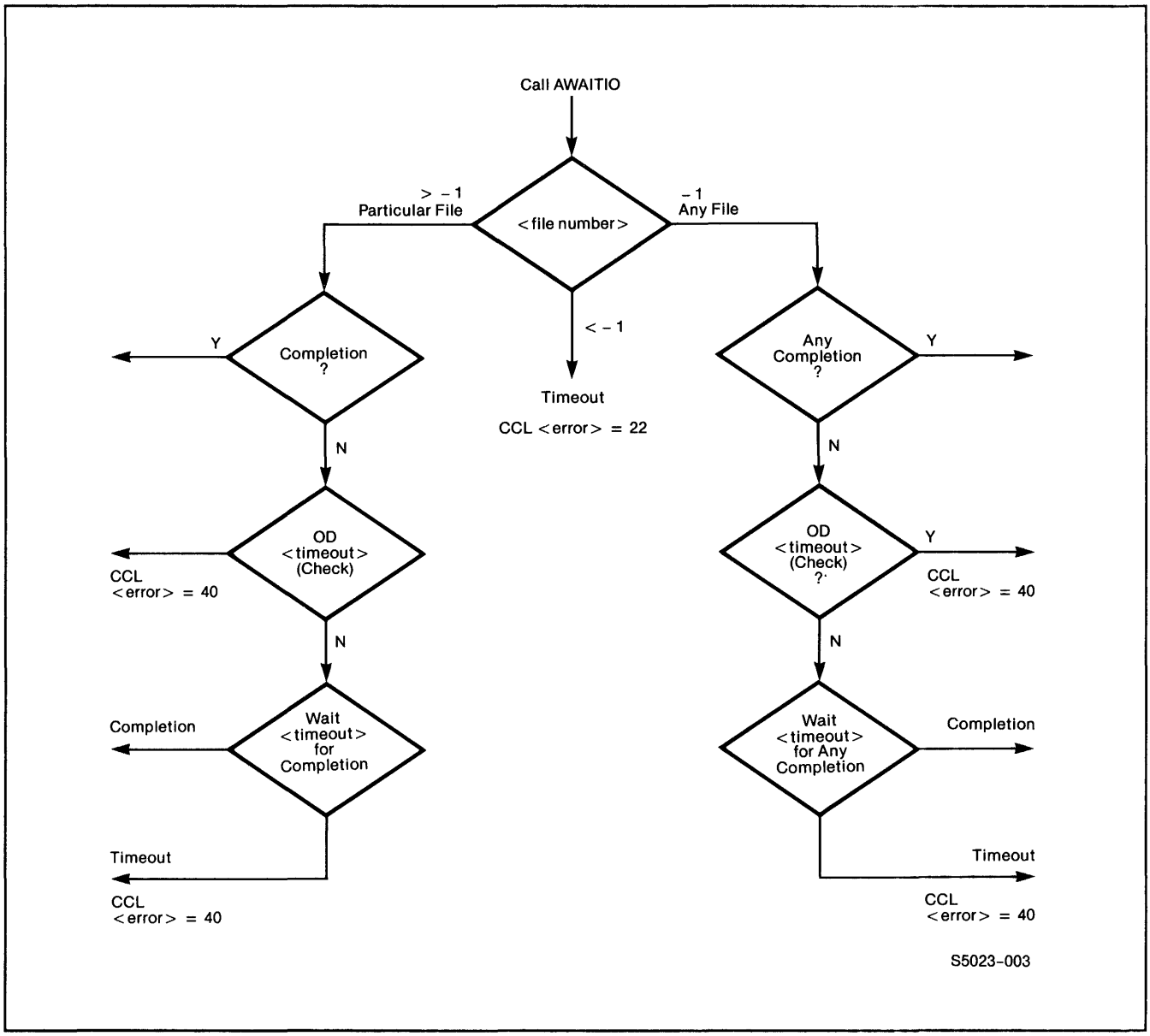


Figure 2-2. AWAITIO Operation

Example

```
CALL AWAITIO ( TERM^NUM , BUFFER , NUM^READ , TAG , FIVE^MINUTES );
```

Related Programming Manual

For programming information about the AWAITIO file system procedure, refer to the GUARDIAN Operating System Programmer's Guide.

BEGINTRANSACTION PROCEDURE

BEGINTRANSACTION starts a new transaction. When you call this procedure, the Transaction Monitoring Facility (TMF) creates a new transaction identifier that becomes the current-transaction identifier for the process issuing BEGINTRANSACTION.

The syntax for BEGINTRANSACTION is:

```
<status>  := BEGINTRANSACTION ( <trans-begin-tag> ) ;     !o
```

<status> returned value

INT

returns a zero, if the call succeeds, or a file system error number. (Refer to the System Messages Manual for a list of all file system errors.)

<trans-begin-tag> output

INT(32):ref

returns a value that identifies the new transaction identifier among other transaction identifiers that the calling process pair has began. NonStop process pairs and processes that have multiple concurrent active transactions require this parameter.

Condition Code Settings

The condition code has no meaning following a call to BEGINTRANSACTION.

Considerations

- Transaction Identifier

  Each transaction is distinguished from other transactions by a four-word transaction identifier, which is created when you

call 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 you call BEGINTRANSACTION. This is 0 for a system that is not part of a network. The system number identifies the home node of the transaction.
.<8:15>	contains the number of the processor in which BEGINTRANSACTION originated.
transid[1:2]	contains a double-word sequence number to make the transaction identification unique.
transid[3]	contains a "crash count" indicating the number of times the home node (of the transaction) has a total system failure since the last time the TMFCOM command INITIALIZE TMF is issued on the home node.

- Restoring a Transaction Identifier

  The value returned to the <trans-begin-tag> can be passed to the RESUMETRANSACTION procedure to restore to currency a transaction identifier previously begun by this process (or its backup). See the explanation of RESUMETRANSACTION in the Transaction Monitoring Facility (TMF) Reference Manual for more details on this operation.

- Sequence-Number Counter of the Processor

  When BEGINTRANSACTION is executed, it increments the sequence-number counter of the processor in which it executes. The value of the sequence-number counter is placed in transaction identifier <transid>[1:2]. See the Transaction Monitoring Facility (TMF) Reference Manual for a description of the sequence-number counter.

- Out-of-Bounds Parameter or Buffer Address

  If you specify an out-of-bounds application parameter or buffer address parameter--that is, a pointer to the buffer has an address that is greater than the MEM associated with the data area of the process--then the call returns with file system error 22.

- Obtaining Main Memory Space For a Link Control Block (LCB)

  If the procedure failed to obtain an LCB, then BEGINTRANSACTION returns file system error 30.

- When BEGINTRANSACTION Fails

  BEGINTRANSACTION fails if TMF is not running on the local system, or the remote system that you try to access. In either case, the call returns file system error 82.

- Too Many Transactions

  If a process begins more concurrent transactions than it can handle (that is, the OPENs against the transaction pseudofile (TFILE) exceed that specified when TFILE was opened) or it attempts more than one open against a TFILE that is not open, the call is rejected with file system error 83. You can have no more than 100 OPENs against the TFILE.

- If TMF Is Not Configured

  If you attempt BEGINTRANSACTION on a system where TMF is not configured, the call fails and error 84 returns.

Example

```
STATUS := BEGINTRANSACTION ( TRANS^BEGIN^TAG );
```

Related Programming Manual

For programming information about the BEGINTRANSACTION procedure, refer to the Transaction Monitoring Facility (TMF) Reference Manual.

BLINK^SCREEN PROCEDURE

For users of the ENTRY or ENTRY520 screen formatter, BLINK^SCREEN places control characters into the application program's I/O buffer, which causes certain characters to blink when appearing on a terminal screen, or which clears the blinking on a data-entry field. When output, this control sequence leaves the cursor over the first character of the field that is blinking or not blinking.

The syntax for BLINK^SCREEN is:

```
<contrl-chars> := BLINK^SCREEN ( @<screen-name>          ! i
                               ,SCREEN                   ! o
                               ,<buffer>                 ! i
                               ,<field-name>             ! i
                               ,<blink>  );              ! i
```

<contrl-chars> returned value

INT

returns the number of control characters that are placed into the application program's I/O buffer.

<screen-name> input

INT:value

is the address of the read-only array that has the form definition (refer to the ENTRY Screen Formatter Operating and Programming Manual or the ENTRY520 Screen Formatter Operating and Programming Manual for an explanation of "form definition").

SCREEN output

STRING:ref:*

is the required array named SCREEN where the entry data is placed. All fields are null-terminated; so, you can access them using the Transaction Application Language (TAL) SCAN statement. Each field is without leading and trailing blanks: in other words, each field is left-justified.

<buffer> input

STRING:ref:*

is the application program's I/O buffer where the control sequence is placed. If you want an entire field to blink, then the control sequence is 20 characters long. To stop a field from blinking, you must specify 18 control-sequence characters.

<field-name> input

STRING:ref:*

is the name of the entry field you want to blink or stop blinking.

<blink> input

INT:value

is nonzero to cause an entry field to blink or zero to stop the entry field from blinking.

Condition Code Settings

The condition code has no meaning following a call to BLINK^SCREEN.

Example

```
NUM^CHARS := BLINK^SCREEN ( @X , SCREEN , BUF , X^NAME , 1 );
```

Related Programming Manuals

For programming information about the BLINK^SCREEN entry procedure, refer to the ENTRY Screen Formatter Operating and Programming Manual or the ENTRY520 Screen Formatter Operating and Programming Manual.

CANCEL PROCEDURE

The CANCEL procedure is used to cancel the oldest incomplete operation on a file opened nowait.

NOTE

You can cancel a specific call, identified with a <tag> parameter, using a call to CANCELREQ.

The syntax for CANCEL is:

```
CALL CANCEL ( <filenum> );                          ! i

<filenum>                    input

  INT:value

  is the number of an open file whose oldest incomplete
  operation you want to cancel.
```

Condition Code Settings

< (CCL) indicates that an error occurred (call FILEINFO).

= (CCE) indicates that the operation was canceled.

> (CCG) does not return from CANCEL.

Example

```
CALL CANCEL ( SOME^FILE );  ! the operation on this file is to be
                            ! canceled.
```

Related Programming Manual

None

CANCELPROCESSTIMEOUT PROCEDURE

The CANCELPROCESSTIMEOUT procedure cancels a process-time timer previously initiated by a call to the SIGNALPROCESSTIMEOUT procedure.

The syntax for CANCELPROCESSTIMEOUT is:

```
CALL CANCELPROCESSTIMEOUT ( <tag> );              ! i

<tag>                          input

  INT:value

  is the identifier associated with the timer to be canceled
  or -1 if all timers set by calls to SIGNALPROCESSTIMEOUT by
  that process are to be canceled.
```

Condition Code Settings

<	(CCL)	is not returned by CANCELPROCESSTIMEOUT.
=	(CCE)	indicates that CANCELPROCESSTIMEOUT was successful.
>	(CCG)	indicates that <tag> was invalid.

Considerations

- The CANCELPROCESSTIMEOUT procedure measures the time the process is executing. This procedure includes only the time spent in process code.

Example

```
CALL CANCELPROCESSTIMEOUT ( TIMERTAG );
```

Related Programming Manual

For programming information about the CANCELPROCESSTIMEOUT procedure, refer to the GUARDIAN Operating System Programmer's Guide.

CANCELREQ PROCEDURE

The CANCELREQ procedure is used to cancel an incomplete operation, identified by a file number and tag, on a nowait file.

The syntax for CANCELREQ is:

```
CALL CANCELREQ ( <filenum>                          ! i
                 [,<tag>]  );                       ! i
```

<filenum> input

INT:value

is the number of an open file, identifying the file whose operation did not complete and is to be canceled.

<tag> input

INT(32):value

is for nowait only. <tag> is a value you define that uniquely identifies the operation associated with this CANCELREQ.

NOTE

The system stores the <tag> value until the I/O operation completes. The system then returns the <tag> information to the program in the <tag> parameter of the call to AWAITIO, thus indicating that the operation was canceled.

Condition Code Settings

< (CCL) does not return from CANCELREQ.

= (CCE) indicates that the operation was canceled.

> (CCG) indicates that an error occurred (call FILEINFO).

Considerations

- Using the <tag> Parameter

  If you use the <tag> parameter, the system cancels the oldest incomplete operation associated with that tag value. If you do not provide a <tag>, the system cancels the oldest incomplete operation for <filenum>.

- If you omit the <tag> parameter, CANCELREQ works exactly like CANCEL.

Example

```
CALL CANCELREQ ( SOME^FILE , 14D );    ! operation 14 of some^file
                                       ! was canceled.
```

Related Programming Manual

For programming information about the CANCELREQ procedure, refer to the GUARDIAN Operating System Programmer's Guide.

CANCELTIMEOUT PROCEDURE

The CANCELTIMEOUT procedure cancels an elapsed-time timer previously initiated by a call to the SIGNALTIMEOUT procedure.

The syntax for CANCELTIMEOUT is:

```
CALL CANCELTIMEOUT ( <tag> );              ! i

<tag>                          input

  INT:value

  is the identifier associated with the timer to be canceled
  or -1 if all timers set by calls to SIGNALTIMEOUT by that process
  are to be canceled.
```

Condition Code Settings

< (CCL) is not returned from CANCELTIMEOUT.

= (CCE) indicates that CANCELTIMEOUT completed successfully.

> (CCG) indicates that <tag> was invalid.

Considerations

- The CANCELTIMEOUT procedure measures the actual elapsed time (the wall clock) that this process executes. This procedure includes the time spent in process code, system code, and interrupt handler code.

Example

```
CALL CANCELTIMEOUT ( TIMERTAG );
```

Related Programming Manual

For programming information about the CANCELTIMEOUT process control procedure, refer to the GUARDIAN Operating System Programmer's Guide.

CHANGELIST PROCEDURE

The CHANGELIST procedure is used only when the application program acts as a supervisor or tributary station in a centralized multipoint configuration.

Within a supervisor station, CHANGELIST performs one of the following operations:

- Specifies continuous or noncontinuous polling
- Enables or disables polling of a particular station
- Resumes polling of partially disabled (that is, nonresponding) stations
- Performs the activation or deactivation of a tributary station by altering the setting of the poll state bit for a particular entry.

NOTE

If polling is in progress when you make the call to CHANGELIST, the specified changes do not take effect until polling completes either on its own or as the result of a call to HALTPOLL.

The syntax for CHANGELIST is:

```
CALL CHANGELIST ( <filenum>                          ! i
                 ,<function>                         ! i
                 ,<parameter> );                     ! i
```

<filenum> input

INT:value

is the name of the one-word integer variable specified in the OPEN call that opened the line.

<function> input

INT:value

is an integer value specifying what change is to be made:

>= 0 changes the poll state bit. In this case, <function> also specifies the relative address of the particular station address within the address list (0 indicates the first entry, 1 the second entry, and so forth). The <parameter> value described under "Considerations" specifies whether you want the bit to be set or cleared.

-1 changes the polling type. The <parameter> value described below specifies whether you want continuous polling or you want the polling list to be traversed a finite number of times.

-2 restores all partially disabled stations.

<parameter> input

INT:value

is an integer value used in conjunction with the <function> value to specify what change is to be made.

<function> >= 0 The <parameter> value specifies whether you want the poll or select state bit set or cleared as follows:

0 = cleared
1 = set

depending upon whether the station list is that of a supervisor or a tributary station:

- Within a supervisor station, the poll state bit enables (clears) or disables (sets) the polling of the particular tributary station.
- Within a tributary station, the poll state bit activates (clears) or deactivates (sets) the tributary station with regard to its ability to respond to a poll or select of the designated station address.

→

<function> = -1	The <parameter> value specifies the desired type of polling as follows: 0 = continuous polling >0 = noncontinuous polling (traverse the polling list the specified number of times, and then cease polling).
<function> = -2	The <parameter> value has no meaning. The CHANGELIST procedure, however, expects to be passed three values; you must therefore supply a dummy <parameter> value.

Condition Code Settings

< (CCL) indicates that an error occurred (call FILEINFO).

= (CCE) indicates that the CHANGELIST procedure executed successfully.

> (CCG) does not return from CHANGELIST.

Examples

```
CALL CHANGELIST ( FNUM , -1 , 10 );
```

In the above example, within a supervisor station, this call enables limited polling in which the station list is traversed 10 times. Polling does not begin, however, until a READ call is subsequently issued. After the tenth pass through the polling list, polling ceases.

Related Programming Manuals

For programming information about the CHANGELIST system procedure, refer to the data-communication manuals.

CHECKCLOSE PROCEDURE

The CHECKCLOSE procedure is called by a primary process to close a designated file in its backup process.

The backup process must be in the "monitor" state (that is, in a call to CHECKMONITOR) for the CHECKCLOSE to be successful. The call to CHECKCLOSE causes the CHECKMONITOR procedure in the backup process to call the file system CLOSE procedure for the designated file.

The syntax for CHECKCLOSE is:

```
CALL CHECKCLOSE ( <filenum>                          ! i
                 ,[ <tape-disposition> ]  );         ! i
```

<filenum> input

INT:value

is the file number of an open file to be closed in the backup process.

<tape-disposition> input

INT:value

if present, specifies mag tape disposition, as follows:

<tape-disposition>.<13:15>

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

If omitted, 0 is used.

Condition Code Settings

The following settings are obtained from the CLOSE procedure in the backup process; CHECKCLOSE establishes these settings in the primary process:

< (CCL) indicates that an invalid file number was supplied or that the backup process does not exist.

= (CCE) indicates that the CLOSE was successful.

> (CCG) does not return from CHECKCLOSE.

Considerations

- Interprocess Message and the Creator Process ID (PID)

  A call to CHECKCLOSE causes an interprocess message to be sent to the process indicated by the "creator PID" in the caller's process control block.

  The creator PID is automatically set to the PID of the backup process at process creation if the NonStop process pair is named. If the process pair is not named, then the backup process must call the STEPMOM procedure, specifying the primary process, before the primary process makes a call to this procedure. (CHECKMONITOR receives and processes the interprocess message in the backup process.)

- The condition code returned from CHECKCLOSE indicates the outcome of the CLOSE in the backup process.

- See "Considerations" for the CLOSE procedure.

Example

```
CALL CHECKCLOSE ( TAPE^FILE , 1 );
```

Related Programming Manual

For programming information about the CHECKCLOSE utility procedure, refer to the GUARDIAN Operating System Programmer's Guide.

CHECKMONITOR PROCEDURE

The CHECKMONITOR procedure is called by a backup process to monitor the state of the primary process and to return control to the appropriate point (in the backup process) in the event of a failure of the primary process.

The syntax for CHECKMONITOR is:

```
{ <status> := } CHECKMONITOR;
{ CALL       }


<status>                        returned value

  INT

  returns a status word of the following form:

    <0:7>   = 2
    <8:15> = 0  primary stopped
             1  primary abnormally ended
             2  primary's processor failed
             3  primary called CHECKSWITCH
```

NOTE

The normal return from a call to CHECKMONITOR is to the statement following a call to CHECKPOINT. The return corresponds to the latest call to CHECKPOINT by the primary process in which its stack was checkpointed.

The backup process executes the statement following the call to CHECKMONITOR only if the primary process has not checkpointed its stack through a call to CHECKPOINT.

Condition Code Settings

The condition code has no meaning following a call to CHECKMONITOR (see the <status> parameter).

Considerations

- Action if the Process Pair Is not Named

  If the process pair is not named (that is, it is not in the destination control table (DCT)), you must call the STEPMOM procedure prior to the call to CHECKMONITOR and before the primary process makes its first call to CHECKPOINT.

- Illegal Parameter Error

  While CHECKMONITOR executes, its local data area consists of approximately 500 words, starting at:

  ```
  'G' [ $LMIN ( LASTADDR, 32767 ) - 500 ]
  ```

  In other words, the local data area begins 500 words below the last available location in the application process's data stack. CHECKMONITOR uses this 500-word region to call other operating system procedures. If the primary process attempts to checkpoint its data area in this region, an "illegal parameter" error returns to the primary process from CHECKPOINT.

  If this failure occurs, the number of data pages to be allotted to the process should be increased through the "?DATAPAGES" Tandem Application Language (TAL) compiler command. (Use this method of increasing data area size rather than increasing the data area at run time through the command interpreter MEM parameter; this helps to avoid creating a backup process with a different data area size than its primary.)

Example

```
CASE CHECKMONITOR OF
  BEGIN
    .
    .
    .
  END;
```

Related Programming Manual

For programming information about the CHECKMONITOR checkpointing procedure, refer to the GUARDIAN Operating System Programmer's Guide.

CHECKOPEN PROCEDURE

The CHECKOPEN procedure is called by a primary process to open a designated file for its backup process. The following two conditions must apply before the call to CHECKOPEN:

- The primary process must first open the file.
- The backup process must be in the "monitor" state (that is, in a call to CHECKMONITOR) for the CHECKOPEN to be successful.

The call to CHECKOPEN causes the CHECKMONITOR procedure in the backup process to call the file system OPEN procedure for the designated file.

The syntax for CHECKOPEN is:

```
CALL CHECKOPEN ( <filename>                                  ! i
               ,<filenum>                                    ! i, o
               ,[ <flags> ]                                  ! i
               ,[ <sync or receive-depth> ]                  ! i
               ,[ <sequential-block-buffer> ]                ! i
               ,[ <buffer-length> ]                          ! i
               ,<backerror> );                               ! o
```

The following parameters must be passed the same values as those passed for the corresponding parameters in the call to OPEN for a file. Otherwise, your program is incorrect.

CHECKOPEN parameter	Corresponding OPEN parameter
<filename>, INT:ref:12	<filename>
<filenum>, INT:value	<filenum>
<flags>, INT:value	<flags>
<sync or receive-depth> INT:value	<sync or receive-depth>
<sequential-block-buffer> INT:ref:*	<sequential-block-buffer>
<buffer-length>, INT:value	<buffer-length>

→

<backerror>	output
INT:ref:1	
returns one of the following values:	
>= 0	is the file system error number reflecting the call to OPEN in the backup process.
-1	indicates that the backup process is not running or that the checkpoint facility could not communicate with the backup process. (See "Messages.")

Condition Code Settings

The following settings are obtained from the OPEN procedure in the backup process:

- < (CCL) indicates that the OPEN failed. The file system error number returns in <backerror>.
- = (CCE) indicates that the file opened successfully.
- > (CCG) indicates that the OPEN was successful, but an exceptional condition was detected. The file system error number returns in <backerror>.

Considerations

- Interprocess Message and the Creator Process ID (PID)

  A call to CHECKOPEN causes an interprocess message to be sent to the process indicated by the "creator PID" in the caller's process control block.

  The creator PID is automatically set to the PID of the backup process at process creation if the NonStop process pair is named. If the process pair is not named, then the backup process must call the STEPMOM procedure, specifying the primary process, before the primary process makes a call to this procedure. (CHECKMONITOR receives and processes the interprocess message in the backup process.) Refer to the GUARDIAN Operating System Programmer's Guide for information about nonnamed process pairs.

- The condition code returned from CHECKOPEN indicates the outcome of the OPEN in the backup process.

- If an Error or No Error Is Returned in <backerror>

  If a process file is opened nowait (<flag>.<8> = 1), that file is CHECKOPENed as nowait. CHECKOPEN returns in <backerror>, errors detected in parameter specification and system data-space allocation, and the operation is considered complete

  If no error returns in <backerror>, the operation must be completed by a call to AWAITIO in the primary process. If you specify the <tag> parameter, the value returned by AWAITIO is -29D; the returned count and buffer address are undefined. If the condition code CCL is returned by AWAITIO, the file is automatically checkclosed by the checkpointing facility. For a nonprocess file or a process file that is opened with a wait, bit <8> of the <flag> parameter is reset internally to zero and ignored. The user can call AWAITIO to complete CHECKOPENs completed for the primary open of the file.

- Primary Process Open

  <backerror> = 17 returns if the file is not opened by the primary process or the parameters supplied to CHECKOPEN do not match the parameters supplied when the primary process opened the file.

- See "Considerations" for the OPEN procedure.

Message

- Unable to Communicate With Backup

  If an "unable to communicate with backup" error occurs (that is, <backerror> = -1), this normally indicates either that the backup process does not exist, or that a system resource problem exists. If a system resource problem exists the open request message to the backup is unduly large.

Example

```
CALL CHECKOPEN ( FNAME1 , FNUM1 , FLAGS1 , SYNC^DEPTH1
                 , , , ERROR );
```

Related Programming Manual

For programming information about the CHECKOPEN checkpointing procedure, refer to the GUARDIAN Operating System Programmer's Guide.

CHECKPOINT PROCEDURE

The CHECKPOINT procedure is called by a primary process to send information pertaining to its current executing state to its backup process. The checkpoint information enables the backup process to recover from a failure of the primary process in an orderly manner. The backup process must be in the "monitor" state (that is, in a call to CHECKMONITOR) for the CHECKPOINT to be successful.

The syntax for CHECKPOINT is:

```
{ <status> := } CHECKPOINT
{ CALL        }

  ( [ <stack-base> [, [ <buffer-1> ] , [ <count-1> ] ]    !i, i, i
                   [, [ <buffer-2> ] , [ <count-2> ] ]    !i, i
                              .              .
                              .              .
                              .              .
                   [, [ <buffer-13>] , [ <count-1 >] ] ); !i, i


<status>                      returned value

  INT

  returns a status word of the following form:

    <0:7> = 0  no error

    <0:7> = 1  no backup or unable to communicate with backup,
               then
      <8:15> = file system error number

    <0:7> = 2  takeover from primary, then
      <8:15> = 0  primary stopped
             = 1  primary abnormally ended
             = 2  primary's processor failed
             = 3  primary called CHECKSWITCH
```

⟶

```
    <0:7> = 3  illegal parameter, then
      <8:15> = number of parameter in error (leftmost position
               = 1)

                          NOTE

       If the message is too large (that is, the stack size
       and the counts of all buffers and the size of all
       file sync blocks are too big), the parameter number
       is set to 13 in the error return.


<stack-base>                  input

  INT:ref:*

  checkpoints the process's data stack from <stack-base> through
  the current top-of-stack location ('S').  A checkpoint of the
  data stack defines a restart point for the backup process.


<buffer-n>                    input

  INT:ref:*

  checkpoints a block of the process's data area (usually a file
  buffer) from <buffer-n> for the number of words specified by
  the corresponding <count-n> parameter. If you omit <buffer-n>,
  <count-n> is treated as a <filenum>, and that file's file
  synchronization block is checkpointed.


<count-n>                     input

  INT:value

  The use of this parameter depends on the presence or absence
  of the corresponding <buffer-n> parameter.

  If <buffer-n> is present, then <count-n> specifies the number of
  words to be checkpointed.

  If <buffer-n> is absent, then <count-n> is the <filenum> of a
  file whose synchronization block is to be checkpointed.
```

Condition Code Settings

The condition code has no meaning following a call to CHECKPOINT.

Considerations

- Checkpointing the Process's Data Stack

  The CHECKPOINT procedure provides for checkpointing the process's data stack and any combination of up to 13 separate data blocks and file synchronization blocks. A data block can be from any location in the data area. (Data blocks are usually file buffers that are not checkpointed as part of the stack, and they cannot be in an extended data area.)

- Interprocess Message and the Creator Process ID (PID)

  A call to CHECKPOINT causes an interprocess message to be sent to the process indicated by the "creator PID" in the caller's process control block.

  The creator PID is automatically set to the PID of the backup process at process creation if the NonStop process pair is named. If the process pair is not named, the backup process must call the STEPMOM procedure, specifying the primary process, before the primary process makes a call to this procedure. (CHECKMONITOR receives and processes an interprocess message in the backup process.)

- Checkpointing a File's Synchronization (sync) Information

  If a file's sync information is checkpointed, the call to CHECKPOINT contains an implicit call to GETSYNCINFO for the file. Therefore, checkpointing of a file's sync information should not be performed between an I/O completion and a call to FILEINFO for that file. If file sync information checkpointing is performed, FILEINFO returns (in its <error> parameter) the status of the call to GETSYNCINFO (usually, <error> = 0).

- Unable to Communicate With Backup

  If an "unable to communicate with backup" error (that is, <status> = 1) occurs, this normally indicates either that the backup process does not exist, or that a system resource problem exists. If the latter, either the checkpoint message to the backup is unduly large, or the SHORTPOOL size in the processor module where the error occurs is too small.

- Illegal Parameter

  If you attempt to checkpoint the data area in the region used by CHECKMONITOR in the backup process, then CHECKPOINT returns an "illegal parameter" error (that is, <status> = 3). See the recovery procedure under "Considerations" for CHECKMONITOR.

Example

```
CALL CHECKPOINT ( STK , , FNUMA );
```

Related Programming Manual

For programming information about the CHECKPOINT checkpointing procedure, refer to the GUARDIAN Operating System Programmer's Guide.

CHECKPOINTMANY PROCEDURE

The CHECKPOINTMANY procedure (like the CHECKPOINT procedure) is called by a primary process to send information pertaining to its current executing state to its backup process.

The CHECKPOINTMANY procedure is used in place of CHECKPOINT when there are more than 13 pieces of information to be sent.

The syntax for CHECKPOINTMANY is:

```
{ <status> := } CHECKPOINTMANY ( [ <stack-base> ]          ! i
{ CALL        }                  ,[ <descriptors> ] );    ! i
```

<status> returned value

INT

returns a status word, in the following form, and one of the following values:

```
<0:7> = 0  no error

<0:7> = 1  no backup or unable to communicate with backup,
           then
  <8:15> = file system error number.

<0:7> = 2  takeover from primary, then
  <8:15> = 0  primary stopped
           1  primary abnormally ended
           2  primary's processor failed
           3  primary called CHECKSWITCH

<0:7> = 3  illegal parameter, then
  <8:15> = 1          error in <stack-base> parameter
         = n, n > 1   error in <word>[n-2] (see
                      "Considerations").
```

<stack-base> input

INT:ref:*

checkpoints the process's data stack from <stack-base> through the current top-of-stack location ('S'). A checkpoint of the data stack defines a restart point for the backup process.

→

<descriptors>	input
INT:ref:*	
is an array that describes the items (data blocks or file synchronization blocks) to be checkpointed. The first word of the array, <descriptors>[0], is a count of the number of items to be checkpointed. <descriptors>[0] is in the range {1:32767}. The rest of the array consists of pairs of words, each pair describing one of the items. (See "Considerations.")	

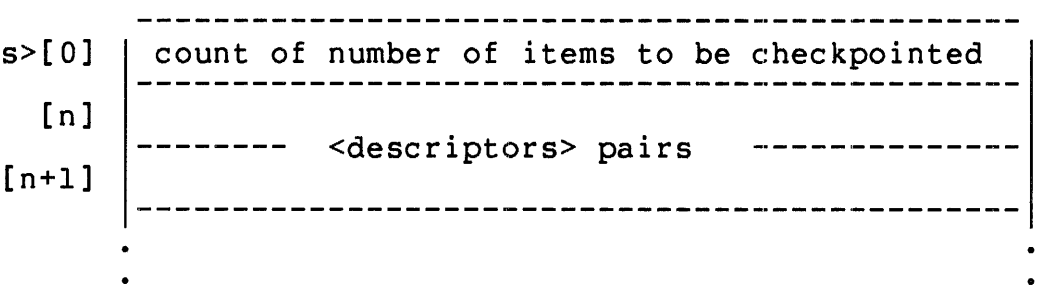
Condition Code Settings

The condition code has no meaning following a call to CHECKPOINTMANY.

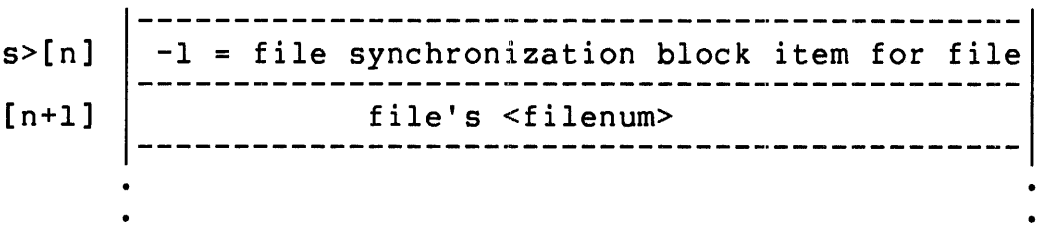
Considerations

- <descriptors> Array Form

  Following word zero, <descriptors> consist of pairs of words.

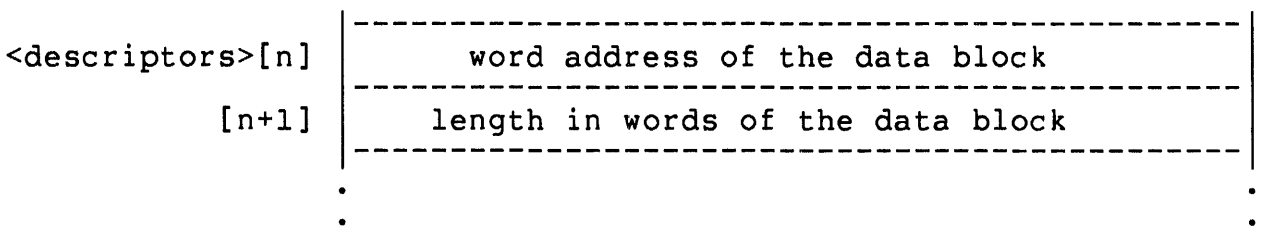


  If the first word of the pair is -1, the pair describes a file synchronization block item for the file whose file number is in the second word of the pair.



  <descriptors> pair[n] = -1
  <descriptors> pair[n+1] = <filenum>

Otherwise, the pair of words describes a data block to be checkpointed: the first word is the word address of the data block, and the second word is the length, in words, of the data block:



<descriptors> pair[n] = <buffer>
<descriptors> pair[n+1] = <count>

The size, in words, of the <descriptors> array must be at least:

1 + 2 * <descriptors>[0].

- <status>.<0:7> = 3, then <status>.<8:15> has the following meaning:

  <status>.<8:15> = 1 error in <stack-base> parameter
  .<8:15> = n, n > 1 error in <word>[n-2]

If the <descriptors> pair describes a file synchronization block (first word of pair = -1, second word = file number), then <descriptors>[n-2] is the second word of the pair in the event of an error (such as GETSYNCINFO failed).

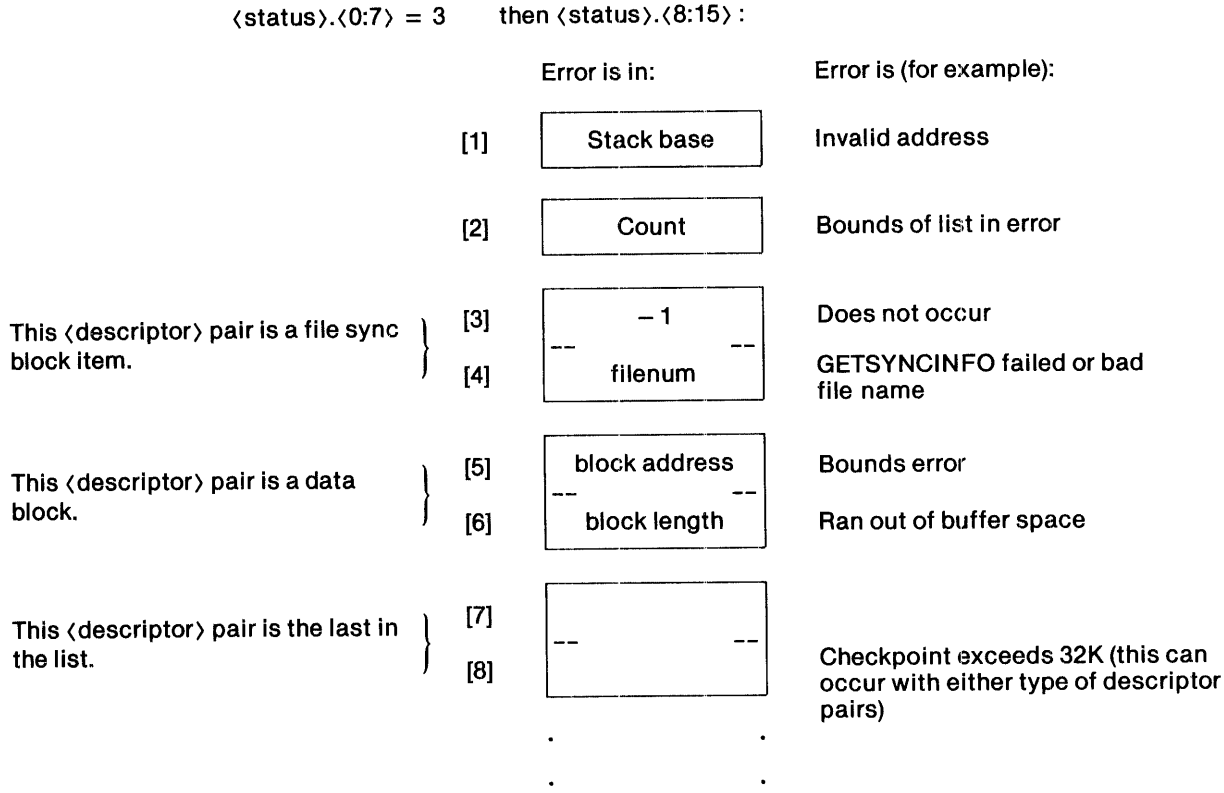
If the pair describes a buffer (first word = address, second word = length), then:

--If the address, or the address plus the length, results in a bounds violation, then <descriptors>[n-2] is the first word of the pair.

--If the pair causes the system to run out of buffer space for the checkpoint, then <descriptors>[n-2] is the second word of the pair.

--If the total amount of data to be checkpointed (data + sync blocks + stack) exceeds 32K bytes, n is set to 2 * <descriptors>[0] + 2.

For example:



- Checkpointing the Process's Data Stack

  The CHECKPOINTMANY procedure allows checkpointing of both the process's data stack and any number of blocks.

  NOTE

  The number of separate data and file synchronization blocks that can be specified is limited by system limits on the size of the resulting message.

- Interprocess Message and the Creator Process ID (PID)

  A call to CHECKPOINTMANY causes an interprocess message to be sent to the process indicated by the "creator PID" in the caller's process control block.

  The creator PID is automatically set to the PID of the backup process at process creation if the primary and backup process pair is named. If the process pair is not named, then the backup process must call the STEPMOM procedure, specifying the primary process, before the primary process makes a call to this procedure. (CHECKMONITOR receives and processes the interprocess message in the backup process.)

- Illegal Parameter

  If an attempt is made to checkpoint the data area used by CHECKPOINTMANY for system-oriented stack maintenance, an "illegal parameter" error (that is, <status> = 3) returns.

- See "Considerations" for the CHECKPOINT procedure.

Example

```
DESCRIPTORS[0] := 2;          ! count of items.
DESCRIPTORS[1] := -1;         ! sync item.
DESCRIPTORS[2] := FNUM^A;     ! file number.
DESCRIPTORS[3] := @BUFFER;    ! data item: word address.
DESCRIPTORS[4] := 512;        ! number of words.
 STAT:= CHECKPOINTMANY( STK^BASE , DESCRIPTOR);
    !  this is equivalent to:
    !  STAT := CHECKPOINT( STK^BASE , , FNUM^A , BUFFER , 512 );
```

Related Programming Manual

For programming information about the CHECKPOINTMANY checkpointing procedure, refer to the GUARDIAN Operating System Programmer's Guide.

CHECK^SCREEN PROCEDURE

For users of the ENTRY or ENTRY520 screen formatter, the CHECK^SCREEN procedure is called after the program inputs the entry-field data from the terminal. The entry fields move out of the application program's I/O buffer into the array SCREEN. Each field is checked, one at a time, according to its data attribute; the checking procedure then performs further checking.

The syntax for CHECK^SCREEN is:

```
<error> := CHECK^SCREEN ( @<screen-name>              ! i
                         ,SCREEN                      ! o
                         ,<buffer>                    ! i
                         ,<check-procedure>           ! i
                         ,<count>  );                 ! o
```

<error> returned value

INT

returns the last value returned by your <check-procedure>.

1 = no errors detected
0 = there was at least one error.

<screen-name> input

INT:value

is the address of the read-only array that has the form definition (refer to the ENTRY Screen Formatter Operating and Programming Manual or the ENTRY520 Screen Formatter Operating and Programming Manual for an explanation of "form definition").

SCREEN output

STRING:ref:*

is the required array named SCREEN where the entry data is placed. All fields are null-terminated; so, you can access them using the Tandem Application Language SCAN statement. Each field is without leading and trailing blanks; in other words, each field is left-justified.

⟶

<buffer> input

STRING:ref:*

is the I/O buffer that holds the data entries as they are when the terminal transmitted them to your program.

<check-procedure> input

INT PROC

is your data-checking procedure. It is called for each field to do further checking.

If it detects an error, then this procedure can abort the checking and provide an application-dependent diagnostic (for the form of the <check-procedure>, see "Considerations").

<count> output

INT:value

is the valid character count in <buffer>. CHECK^SCREEN only processes <count> characters. This value is normally the "count read" parameter returned by the WRITEREAD call following the call to READ^SCREEN.

Condition Code Settings

The condition code has no meaning following a call to CHECK^SCREEN.

Considerations

- The <check-procedure> Form

  Declare the procedure as follows:

```
INT PROC <check-procedure> ( <fieldnum> , <field> , <check>
                                 , <error> );
```

  <fieldnum>

  INT:value

  is the number of the field; fields are numbered starting at 1.

<field>

STRING:ref:*

is a pointer to the entry field in SCREEN array.

<check>

INT:value

is a data attribute defined when the form is created.

<error>

INT:value

is one of the following values:

1 = CHECK^SCREEN detected an error in this field
0 = no error detected

Your <check-procedure> returns a 1 if the checking is to continue or a 0 if it is not.

- The General Control Flow of CHECK^SCREEN

  For each field from the screen:

  --Move the field-entry data into the SCREEN array.
  --Do type checking depending on a field's data attribute.
  --Call your procedure; 0 returns if your procedure finds an error.
  --Return a 1.

  In the case of a zero, the program can redisplay the bad field with the blinking feature, which also places the cursor at the first character of the bad data.

Example

```
ERROR := CHECK^SCREEN ( @X , SCREEN , BUF , CHECKX );
```

Related Programming Manuals

For programming information about the CHECK^SCREEN entry procedure, refer to the ENTRY Screen Formatter Operating and Programming Manual or the ENTRY520 Screen Formatter Operating and Programming Manual.

CHECKSWITCH PROCEDURE

The CHECKSWITCH procedure is called by a primary process to cause the duties of the primary and backup processes to be interchanged.

The call to CHECKSWITCH contains an implicit call to CHECKMONITOR, so that the caller becomes the backup and monitors the execution state of the new primary. The backup process must be in the "monitor" state (that is, in a call to CHECKMONITOR) for the CHECKSWITCH to be successful.

The syntax for CHECKSWITCH is:

```
{ <status> := } CHECKSWITCH;
{ CALL       }


<status>                        returned value

  INT

  returns a status word of the following form with one of the
  following values:

    <0:7> = 1  could not communicate with backup, then
      <8:15> = file system error number.

    <0:7> = 2  then
      <8:15> = 0  primary stopped
             = 1  primary abnormally ended
             = 2  primary's processor failed
             = 3  primary called CHECKSWITCH
```

NOTE

The normal return from a call to CHECKSWITCH is to the statement following a call to CHECKPOINT. The return corresponds to the latest call to CHECKPOINT by the primary process in which its stack is checkpointed.

The backup process executes the statement following the call to CHECKSWITCH only if the primary process has not checkpointed its stack through a call to CHECKPOINT.

Condition Code Settings

The condition code has no meaning following a call to CHECKSWITCH.

Considerations

- When to Use CHECKSWITCH

  Use CHECKSWITCH following the reload of a processor module. The purpose is to switch the process pair's work back to the original primary processor module. CHECKSWITCH causes the current backup to become the primary process and to begin processing from the latest call to CHECKPOINT.

- Interprocess Message and the Creator Process ID (PID)

  A call to CHECKSWITCH causes an interprocess message to be sent to the process indicated by the "creator PID" in the caller's process control block.

  The creator PID is automatically set to the PID of the backup process at process creation if the primary and backup process pair is named. If the process pair is not named, then the backup process must call the STEPMOM procedure, specifying the primary process, before the primary process makes a call to this procedure. (CHECKMONITOR receives and processes the interprocess message in the backup process.)

- For information about the action of CHECKSWITCH following the takeover by the backup process, refer to the GUARDIAN Operating System Programmer's Guide.

Example

```
STAT := CHECKSWITCH;
```

Related Programming Manual

For programming information about the CHECKSWITCH checkpointing procedure, refer to the GUARDIAN Operating System Programmer's Guide.

CLOSE PROCEDURE

The CLOSE procedure is used to close an open file. Closing a file terminates access to the file.

The syntax for CLOSE is:

```
CALL CLOSE ( <filenum>                              ! i
           ,[ <tape-disposition> ] );               ! i
```

<filenum> input

INT:value

is the number of an open file that is to be closed.

<tape-disposition> input

INT:value

is one of the following values, indicating what tape control action to take:

<tape-disposition>.<13:15>

0 = rewind and unload, do not wait for completion
1 = rewind, take offline, do not wait for completion
2 = rewind, leave online, do not wait for completion
3 = rewind, leave online, wait for completion
4 = do not rewind, leave online

Condition Code Settings

< (CCL) indicates that the file was not open.

= (CCE) indicates that the CLOSE was successful.

> (CCG) does not return from CLOSE.

Considerations

- Returning Space Allocation After Closing a File

  Closing a disc file causes the space that is used by the resident file control block to be returned to the system main-memory pool if the disc file is not open concurrently.

  A temporary disc file is purged if the file was not open concurrently. Any space that is allocated to that file is made available for other files.

  With any file closure, the space allocated to the access control block (ACB) is returned to the system.

- Closing a Nowait File

  If a CLOSE is issued for a nowait file that has pending operations, any incomplete operations are canceled. There is no indication as to whether the operation completed or not.

Messages

- Process CLOSE Message

  A CLOSE of a file representing another process can cause a CLOSE system message (that is, system message -31) to be sent to that process.

  A process receives this message when it is closed by another process. You can obtain the <process-id> of the closer in a subsequent call to LASTRECEIVE or RECEIVEINFO. (Refer to Appendix F for detailed information of system messages sent to processes.)

  NOTE

  This message is also received if the close is made by the backup process of a process pair. Therefore, a process can expect two of these messages when being closed by a process pair.

Example

```
CALL CLOSE ( TAPE^FNUM , REW^UNLOAD );
```

Related Programming Manuals

For programming information about the CLOSE file system procedure, refer to the GUARDIAN Operating System Programmer's Guide and the ENSCRIBE Programming Manual.

COMPUTEJULIANDAYNO PROCEDURE

The COMPUTEJULIANDAYNO procedure converts a Gregorian calendar date on or after January 1, 0001, to a Julian day number.

The Julian calendar is the integral number of days since January 1, 4713 B.C. The formal definition of the Julian day number states that it starts at noon, Greenwich mean time (GMT). For simplicity, we assume the Julian day number starts at midnight.

The Gregorian calendar is the common civil calendar that we use today.

The syntax for COMPUTEJULIANDAYNO is:

```
<julian-day-num> := COMPUTEJULIANDAYNO ( <year>                ! i
                                       , <month>               ! i
                                       , <day>                 ! i
                                       ,[ <error-mask> ] );    ! o
```

<julian-day-num> returned value

INT(32)

returns the Julian day number.

<year> input

INT:value

is the Gregorian year (for example, 1984, 1985, ...).
The range for <year> is restricted to 1-4000.

<month> input

INT:value

is the Gregorian month (1-12).

<day> input

INT:value

is the Gregorian day of the month (1-31).

→

<error-mask> output

INT:ref:1

is an array that corresponds (bit by bit) to what was supplied in <year>, <month>, and <day>. If omitted, <year>, <month>, and <day> are not checked.

<error-mask> bits are (starting from the leftmost bit <0>):

.<0> = year
.<1> = month
.<2> = day

If any one of these bits contains a 1, there is an error.

Condition Code Settings

The condition code has no meaning following a call to COMPUTEJULIANDAYNO.

Example

```
JDN := COMPUTEJULIANDAYNO ( YR , MN , DAY , VALIDITY );
```

Related Programming Manual

For programming information about the COMPUTEJULIANDAYNO procedure, refer to the GUARDIAN Operating System Programmer's Guide.

COMPUTETIMESTAMP PROCEDURE

The COMPUTETIMESTAMP procedure converts a Gregorian (common civil calendar) date and time into a 64-bit timestamp.

The syntax for COMPUTETIMESTAMP is:

```
<ret-timestamp> := COMPUTETIMESTAMP ( <date-n-time>          ! i
                                    ,[ <errormask> ]  );    ! o
```

<ret-timestamp> returned value

FIXED

returns a 64-bit timestamp, computed from <date-n-time>.

<date-n-time> input

INT:ref:8

is an array containing a date and the time of day. This array has the following form:

```
<date-n-time>[0] = the Gregorian year (for example, 1984, 1985,
                   ... )
               [1] = the Gregorian month                  (1-12)
               [2] = the Gregorian day of the month       (1-31)
               [3] = the hour of the day                  (0-23)
               [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)
```

The range of the year is restricted to 1-4000.

<errormask> output

INT:ref:1

is an array that corresponds to what was supplied in the <date-n-time> parameter. <errormask> checks each element of <date-n-time> for validity. If omitted, <date-n-time> is not checked.

→

An error is indicated if any of the following bits contain a 1.

<error-mask> bits:

.<0> = year
.<1> = month
.<2> = day
.<3> = hour of day
.<4> = minute of hour
.<5> = second of minute
.<6> = millisecond of second
.<7> = microsecond of millisecond

Condition Code Settings

The condition code has no meaning following a call to COMPUTETIMESTAMP.

Example

```
RETURNED^TIMESTAMP := COMPUTETIMESTAMP ( DATE^N^TIME
                                       , VALIDITY^CHECK );
```

Related Programming Manual

For programming information about the COMPUTETIMESTAMP procedure, refer to the GUARDIAN Operating System Programmer's Guide.

CONTIME PROCEDURE

The CONTIME procedure converts a 48-bit timestamp to a date and time in integer form.

The syntax for CONTIME is:

```
CALL CONTIME ( <date-and-time>              ! o
             ,<t0>                          ! i
             ,<t1>                          ! i
             ,<t2> );                       ! i
```

<date-and-time> output

INT:ref:7

is an array where CONTIME returns a date and time in the following form:

```
<date-and-time>[0] = year    (1975, 1976, ... )
               [1] = month   (1-12)
               [2] = day     (1-31)
               [3] = hour    (0-23)
               [4] = minute  (0-59)
               [5] = second  (0-59)
               [6] = .01 sec (0-99)
```

<t0, t1, t2> input

INT:value:3

is an array that must correspond to the 48 bits of a timestamp for the results of CONTIME to have any meaning (<t0> is the most significant word: <t2> is the least significant).

<t0>	most significant word, interval clock
<t1>	interval clock
<t2>	least significant word, interval clock

Condition Code Settings

The condition code has no meaning following a call to CONTIME.

Example

```
CALL CONTIME( DATE^TIME , LAST^T , LAST^T[1] , LAST^T[2] ) ;
       !conversion to date and time
```

CONTIME is used to convert the three words in LAST^T to a date and time. DATE^TIME returns seven words of date and time.

Related Programming Manual

For programming information about the CONTIME utility procedure, refer to the GUARDIAN Operating System Programmer's Guide.

CONTROL PROCEDURE

The CONTROL procedure is used to perform device-dependent I/O operations.

The syntax for CONTROL is:

```
CALL CONTROL ( <filenum>                              ! i
             ,<operation>                             ! i
             ,<param>                                 ! i
             ,[ <tag> ]  );                           ! i
```

<filenum> input

INT:value

is a number of an open file, identifying the file to which the CONTROL procedure is to perform an I/O operation.

<operation> input

INT:value

is a value that defines a device and an operation with that device (see Appendix A for a list of operation numbers).

<param> input

INT:value

is a value (see Appendix A for a description of control parameters).

<tag> input

INT(32):value

applies to nowait I/O only. <tag> is a value you define uniquely identifying the operation associated with this CONTROL.

⟶

NOTE

The system stores the <tag> value until the I/O operation completes. The system then returns the <tag> information to the program in the <tag> parameter of the call to AWAITIO, thus indicating that the operation completed.

Condition Code Settings

< (CCL) indicates that an error occurred (call FILEINFO).

= (CCE) indicates that the CONTROL was successful.

> (CCG) for magnetic tape, indicates that the end of file (EOF) was encountered while spacing records; for a process file, this setting indicates that the process is not accepting CONTROL system messages. When device handlers do not allow the <operation>, file system error 2 returns.

Considerations

- Nowait and CONTROL

  If the CONTROL procedure is used on a file that is opened nowait, it must be completed with a call to the AWAITIO procedure.

- Disc Files

  --Writing EOF to an unstructured file

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

  ```
  end-of-file pointer := next-record pointer;
  ```

  (File pointer action for CONTROL operation 2 (write EOF).)

  --File is locked

  If a CONTROL operation is attempted for a file locked through a <filenum> other than that specified in the call to CONTROL, the call is rejected with a "file is locked" error 73.

--Invalid operation attempted on audited file or nonaudited disc volume

Attempts to purge data (<operation> 20) from files audited by the Transaction Monitoring Facility are rejected with file system error 80.

- Magnetic Tapes

--When device is not ready

If mag tape rewind is performed concurrently with application program execution (that is, rewind operation <> 6), any attempt to perform a read, write, or control operation to the rewinding tape unit while rewind is taking place results in an error indication. A subsequent call to FILEINFO returns error 100.

--Wait for rewind to complete

If mag tape rewind operation = 6 (wait for completion) is performed as a nowait operation, the application waits at the call to AWAITIO for the rewind to complete.

- Interprocess Communication

--Nonstandard <operation> and <parameter> values

Any value can be specified for the <operation> and <parameter> parameters. An application-defined protocol should be established for interpreting nonstandard parameter values.

--Process not accepting system messages

If the object of the control operation is not accepting CONTROL/CONTROLBUF/SETMODE/SETMODENOWAIT system messages, the call to CONTROL completes with a condition code of CCG; a subsequent call to FILEINFO returns error 7.

The process ID (PID) of the caller to CONTROL can be obtained in a subsequent call to LASTRECEIVE or RECEIVEINFO.

--Process control (see "Messages")

Messages

- Process Control

  This message is received when another process calls the CONTROL procedure, referencing the receiver process file. You can obtain the PID of the caller to CONTROL in a subsequent call to LASTRECEIVE or RECEIVEINFO.

Example

```
CALL CONTROL ( FILE^NUM , FORMS^CONT , VFU^CHANNEL );
```

Related Programming Manuals

For programming information about the CONTROL file system procedure, refer to the GUARDIAN Operating System Programmer's Guide, the ENSCRIBE Programming Manual, and the data communication manuals.

CONTROLBUF PROCEDURE

The CONTROLBUF procedure is used to perform device-dependent I/O operations requiring a data buffer.

The syntax for CONTROLBUF is:

```
CALL CONTROLBUF ( <filenum>                          ! i
                 ,<operation>                        ! i
                 ,<buffer>                           ! i
                 ,<count>                            ! i
                 ,[ <count-transferred> ]            ! o
                 ,[ <tag> ] );                       ! i
```

<filenum> input

INT:value

is the number of an open file. It identifies the file on which the CONTROLBUF procedure performs an I/O operation.

<operation> input

INT:value

is a value defined by the device (see Appendix A).

<buffer> input

INT:ref:*

is an array that contains the information to be used for the CONTROLBUF operation.

<count> input

INT:value

is the number of bytes contained in <buffer>.

→

<count-transferred> output

INT:ref:1

returns a count of the number of bytes transferred from <buffer> (for wait I/O only).

<tag> input

INT(32):value

is for nowait I/O only. <tag> is a value you define that uniquely identifies the operation associated with this CONTROLBUF.

NOTE

The system stores the <tag> value until the I/O operation completes. The system then returns the <tag> information to the program in the <tag> parameter of the call to AWAITIO, thus indicating that the operation completed.

Table 2-1 shows the CONTROLBUF operations.

Table 2-1. CONTROLBUF Operations

<operation>	
1	= load DAVFU (printer subtype 4)
	<buffer> = VFU buffer to be loaded
	<count> = number of bytes contained in <buffer>

Condition Code Settings

< (CCL) indicates that an error occurred (call FILEINFO).

= (CCE) indicates that the CONTROLBUF was successful.

> (CCG) for a process file, indicates that the process is not accepting CONTROLBUF system messages.

Considerations

- Wait and <count-transferred>

  If a "wait" CONTROLBUF is executed, the <count-transferred> parameter indicates the number of bytes actually transferred.

- Nowait and <count-transferred>

  If a nowait CONTROLBUF is executed, <count-transferred> has no meaning and can be omitted. A count of the number of bytes transferred is obtained by the <count-transferred> parameter of the AWAITIO procedure when the I/O completes.

  The CONTROLBUF procedure must complete with a call to the AWAITIO procedure when used with a file opened nowait.

- When Object of CONTROLBUF Is Not Accepting Messages

  If the object of the CONTROLBUF operation is not accepting CONTROL, CONTROLBUF, or SETMODE system messages, the call to CONTROLBUF completes with condition code CCG. A subsequent call to FILEINFO returns error 7 (process not accepting CONTROL, CONTROLBUF, or SETMODE messages).

  You can specify the process ID of the caller to CONTROL or CONTROLBUF in a subsequent call to LASTRECEIVE or RECEIVEINFO.

- Interprocess Communication

  --Nonstandard <operation> and <buffer> parameters

    You can specify any value for the <operation> parameter, and you can include any data in <buffer>. An application-defined protocol should be established for interpreting nonstandard parameter values.

  --Process not accepting system messages

    If the object of the control operation is not accepting CONTROL/CONTROLBUF/SETMODE/SETMODENWAIT system messages, the call to CONTROLBUF completes with a condition code of CCG; a subsequent call to FILEINFO returns error 7.

Message

- Process CONTROLBUF Message

  Issuing a CONTROLBUF to a file that represents another process causes a CONTROLBUF system message (-35) to be sent to that process.

Example

```
CALL CONTROLBUF ( PRINTER , LOAD^VFU , VFU^BUFFER , 132
                , COUNT^XFERRED );
```

Related Programming Manual

For programming information about the CONTROLBUF file system procedure, refer to the GUARDIAN Operating System Programmer's Guide.

CONVERTPROCESSNAME PROCEDURE

The CONVERTPROCESSNAME procedure converts a process name from local to network form.

The syntax for CONVERTPROCESSNAME is:

```
CALL CONVERTPROCESSNAME ( <process-name> );                ! i, o

<process-name>                input, output

  INT:ref:3

  is a process name beginning with "$" to be converted.

  On return, <process-name> contains the internal network form
  of the process name:  "\" in the first byte and the calling
  process's system number in the second byte, followed by the
  process name.

  If <process-name> does not begin with "$," it is left unchanged.
```

Condition Code Settings

The condition code has no meaning following a call to CONVERTPROCESSNAME.

Considerations

- When using this call, any process name that is longer than four characters plus the "$" is truncated.

Example

```
NAME ':=' "$PROC";
CALL CONVERTPROCESSNAME ( NAME );
```

Related Programming Manual

For programming information about the CONVERTPROCESSNAME process control procedure, refer to the GUARDIAN Operating System Programmer's Guide.

CONVERTPROCESSTIME PROCEDURE

The CONVERTPROCESSTIME procedure is used to convert the quad microsecond process time returned by the PROCESSTIME, MYPROCESSTIME, or PROCESSINFO procedure into hours, minutes, seconds, milliseconds, and microseconds. The maximum time that this procedure can convert is 3.7 years (this is the amount of time that can be represented using the output parameters).

The syntax for CONVERTPROCESSTIME is:

```
CALL CONVERTPROCESSTIME ( <process-time>              ! i
                        ,[ <hours> ]                  ! o
                        ,[ <minutes> ]                ! o
                        ,[ <seconds> ]                ! o
                        ,[ <milliseconds> ]           ! o
                        ,[ <microseconds> ]  );       ! o
```

<process-time> input

FIXED

specifies the time to be converted.

<hours> output

INT:ref:1

returns the hours portion of the <process-time> specified.

<minutes> output

INT:ref:1

is the minutes portion of the <process-time> specified.

<seconds> output

INT:ref:1

is the seconds portion of the <process-time> specified.

→

<milliseconds>	output
INT:ref:1	
is the milliseconds portion of the <process-time> specified.	
<microseconds>	output
INT:ref:1	
is the microseconds portion of the <process-time> specified.	

Condition Code Settings

<	(CCL)	returns if <process-time> represents a quantity greater than 3.7 years.
=	(CCE)	indicates that CONVERTPROCESSTIME is successful.
>	(CCG)	returns if any of the supplied output parameters fail the bounds check on the address.

Considerations

- If the output parameters are not specified the information that would be returned is lost.

Example

```
CALL CONVERTPROCESSTIME ( PROC^TIME
                        , HOURS
                        ,                  ! minutes.
                        , SECONDS
                        ,                  ! milliseconds.
                        , MICRO^SEC );
```

Related Programming Manual

None

CONVERTTIMESTAMP PROCEDURE

The CONVERTTIMESTAMP procedure converts a Greenwich mean time (GMT) timestamp to or from a local time based timestamp within any accessible node in the network.

A local timestamp can be local standard time ((LST) does not include daylight savings time (DST)), local civil time ((LCT) includes DST).

DST is a system to extend the amount of daylight in the evenings by changing (advancing) the time. Usually, but not always, this standard is done in hour increments. In most states in the United States, DST begins at 2:00 A.M. on the last Sunday in April and ends at 2:00 A.M. LST (3:00 A.M. DST) on the last Sunday in October; the United States advances the time by one hour.

The syntax for CONVERTTIMESTAMP is:

```
<ret-time> := CONVERTTIMESTAMP ( <julian-timestamp>          ! i
                               ,[ <direction> ]              ! i
                               ,[ <node> ]                   ! i
                               ,[ <error> ] );               ! o


<ret-time>                  returned value

  FIXED

  returns a Julian timestamp with a different base.


<julian-timestamp>          input

  FIXED:value

  is a four-word Julian timestamp.


<direction>                 input

  INT:value

  indicates what time form or timestamp to return.  You can
  specify one of the following values for <direction>.
```

→

```
    0 = GMT to local civil time (LCT) (default)
    1 = GMT to local standard time (LST)
    2 = LCT to GMT
    3 = LST to GMT

  If <direction> is out of range, 0 is assumed.


<node>                          input

  INT:value

  is the system number at the node where you want the conversion.
  The default is local.  If the specified <node> does not exist,
  the value of <ret-value> is not changed.  (This parameter is
  valid only for the B00 version of the operating system.)


<error>                         output

  INT:ref:1

  returns one of the following values:

    -1 = ambiguous LCT
    -2 = impossible LCT
     0 = no errors, successful
     1 = DST range error
     2 = DST table not loaded
    >2 = file system error (attempting to reach "NODE")
```

Condition Code Settings

The condition code has no meaning following a call to CONVERTTIMESTAMP.

Considerations

- A local timestamp can be in either of two forms; LCT (with DST correction), or LCT (without DST correction).

- Network and Local Timestamp

  Local timestamp (with LCT and LST) should be used with caution if any network use is anticipated. The reason is that another node can be in another time zone or in an area with different DST rules (LCT only).

- LCT Timestamps

  LCT timestamps should be used with caution because of the negative adjustment that DST systems dictate. Timestamp base conversion (for example, LCT) is provided by the operating system.

Example

```
RETURN^TIME^BASE := CONVERTTIMESTAMP ( JULIAN
                                     ,
                                     , SYSTEM^NUM );
```

Related Programming Manual

None

CPUTIMES PROCEDURE

The CPUTIMES procedure returns the length of time, in microseconds, since cold load that a given CPU has spent in the following states:

- process busy
- interrupt busy
- idle

These times reflect the amount of time spent by the CPU (last cold load or reload) in a process environment, an interrupt environment, and the idle state.

The syntax for CPUTIMES is:

```
CALL CPUTIMES ( [ <cpu> ]                          ! i
              ,[ <sysid> ]                         ! i
              ,[ <total-time> ]                    ! o
              ,[ <cpu-process-busy> ]              ! o
              ,[ <cpu-interrupt> ]                 ! o
              ,[ <cpu-idle> ]  );                  ! o
```

<cpu> input

INT:value

specifies the CPU number of a CPU in the system. The default is the local CPU.

<sysid> input

INT:value

specifies the system number. The default is the local system.

<total-time> output

FIXED:ref:1

returns the elapsed time, in microseconds, since the CPU was cold loaded or reloaded.

→

<cpu-process-busy> output

FIXED:ref:1

returns the length of time, in microseconds, the CPU has been busy executing processes since the last cold load or reload.

<cpu-interrupt> output

FIXED:ref:1

returns the length of time, in microseconds, the CPU has been busy processing interrupts since the last cold load or reload.

<cpu-idle> output

FIXED:ref:1

returns the length of time, in microseconds, the CPU has been idle since the last cold load or reload.

Condition Code Settings

< (CCL) indicates that the system is in one of the following states:

- unavailable
- does not exist
- the procedure could not get resources to execute
- the system is running on an earlier operating system version than B00.

= (CCE) indicates that CPUTIMES is successful.

> (CCG) indicates the supplied parameters failed the bounds check.

Example

```
CALL CPUTIMES ( PROCESSOR , SYS^NUM , COLD^TIME , CPU^BUSY
              , CPU^INTERRUPT, CPU^IDLE );
```

Related Programming Manual

None

CREATE PROCEDURE

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

The syntax for CREATE is:

```
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
```

<filename> input, output

INT:ref:12

is an array containing the name of the disc file to be created. <filename> must be in one of the following forms (to create a permanent or temporary disc file):

Permanent Disc File

```
<filename>[0:3]  is  $<volname><blank-fill>
                              or
                     \<sysnum><volname><blank-fill>
<filename>[4:7]  is   <subvol-name><blank-fill>
<filename>[8:11] is   <disc-filename><blank-fill>
```

Temporary Disc File

<filename>[0:11] is $<volname><blank-fill>
or
\<sysnum><volname><blank-fill>

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

<primary-extentsize> input

INT:value

is the size of the primary extent in pages (one page is 2048-byte). The maximum <primary-extentsize> extent size is 65,535 (134,215,680 bytes). If omitted, a primary extent size of 1 page is assigned.

<file-code> input

INT:value

is an application-defined file identification code (file codes 100-999 are reserved for use by Tandem.) If omitted, a file code of 0 is assigned. Refer to the GUARDIAN Operating System Utilities Reference Manual for a list of Tandem file codes.

<secondary-extentsize> input

INT:value

is the size of the secondary extents in pages (one page is 2048-byte). (A file can have up to 15 secondary extents allocated.) The maximum <secondary-extentsize> is 65535 (134,215,680 bytes). If omitted, the size of the primary extent is used for the secondary extent size.

<file-type> input

INT:value

specifies the type of file to be created. If omitted, an unstructured file is created.

→

<file-type>.<0:1> must be 0.

.<2> in systems with the Transaction Monitoring Facility (TMF), specifies this file is audited; for systems without TMF, this bit is 0.

.<3:9> must be 0.

.<10> specifies that the file label is written to disc each time the end of file (EOF) is advanced. The effect of setting this parameter is the same as calling REFRESH after every operation that advances the EOF pointer.

.<11> specifies index compression for key-sequenced files (see the ENSCRIBE Programming Manual).

.<12> specifies ODDUNSTR access to unstructured files. With the default (<file-type> .<12>=0), a relative byte address (RBA) used for reading, writing, or positioning in the file is rounded up to the next even number (whole word boundary); thus, 3 rounds up to 4, and so forth. ODDUNSTR prevents this rounding, so that reading, writing, or positioning occurs at the exact RBA specified. (See "Considerations.")

.<12> specifies data compression for key-sequenced files (Refer to the ENSCRIBE Programming Manual for additional information).

.<13:15> specifies the file structure:

0 = unstructured (default)
1 = relative
2 = entry-sequenced
3 = key-sequenced.

<recordlen> input

INT:value

is the maximum length of the logical record in bytes.

For DP1 structured files, the maximum <recordlen> is determined by the data-block size. With a data-block size of 4096; the maximum record length for entry-sequenced and relative files is 4072, and 2035 for a key-sequenced files.

For an unstructured file, the maximum record length is 4096.

If omitted, <recordlen> = 80.

The formulas for computing the maximum record size (MRS) based on <blksize> are:

For relative and entry-sequenced files:

MRS = <data-block-size> - 24

For key-sequenced files:

MRS = 1/2 * (<data-block size> - 26)

For unstructured files:

MRS = <data-block-size>

<data-blocklen> input

INT:value

for structured files, is the length in bytes of each block of records in the file. <data-blocklen> must be a multiple of 512 and cannot be greater than 4096. <data-blocklen> must be at least <recordlen> + 24. For a key-sequenced file, <data-blocklen> must be at least 2 * <recordlen> + 26.

If omitted, 1024 is used for the <data-blocklen> (for DP1 disc files). Regardless of the specified record length and data-block size, the maximum number of records that can be stored in a data block is 511.

→

For DP2 the data-block sizes are limited to power-of-two multiples of the sector size; 512, 1024, 2048, and 4096. For example, 3K byte blocks are not supported by DP2. (The default is 4096.)

<key-sequenced-params> input

INT:ref:3

is a three-word array containing parameters that describe this file. This parameter is required for key-sequenced files ,but you can omit the parameter for other file types. See "Considerations" for the format of this array.

<alternate-key-params> input

INT:ref:*

is an array containing parameters describing any alternate keys for this file. This parameter is required if the file has alternate keys; otherwise, you can omit this parameter. If included, the first character must be 0 if you have alternate keys. See "Considerations" for the format of this array.

<partition-params> input

INT:ref:*

is an array containing parameters that describe this file. It applies only if the file is a multivolume file. If the file is to span multiple volumes, this parameter is required; otherwise, you can omit it. If included, the first character must be 0. See "Considerations" for the format of this array.

<maximum-extents> input

INT:value

is the maximum number of extents to be allocated for the file. The default is 16. This parameter is valid for a DP2 disc file only.

→

```
<unstructured-buffer-size>      input

  INT:value

  declares the internal buffer size to be used for an
  unstructured file; valid for DP2 only and must be a
  valid DP2 blocksize.  Valid DP2 blocksizes are 512, 1024, 2048,
  and 4096.  The default is 4096 bytes.


<open-defaults>                 input

  INT:value

  specifies the file label default values for various open
  attributes.  This parameter is valid for a DP2 disc file
  only.

    <open-defaults>.<0> = 0  verify WRITEs off (default)
                        = 1  verify WRITE on

                    .<1> = 0  system automatically selects serial
                              or parallel WRITEs
                         = 1  serial mirror WRITEs only

                    .<2> = 0  buffered WRITEs enabled (default
                              for audited files)
                         = 1  WRITE-thru (default for
                              nonaudited files)

                    .<3> = 0  audit compression off (default)
                         = 1  audit compression on
```

Condition Code Settings

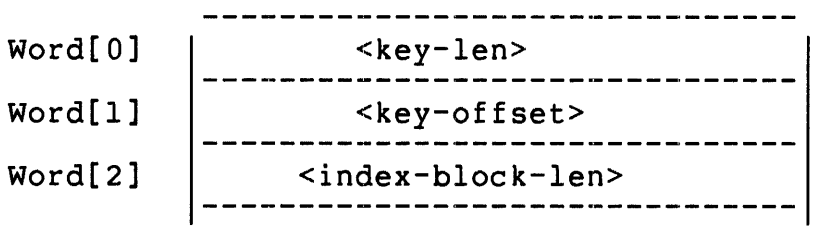
< (CCL) indicates that the CREATE failed (call FILEINFO).

= (CCE) indicates that the file was created successfully.

> (CCG) indicates that the device is not a disc.

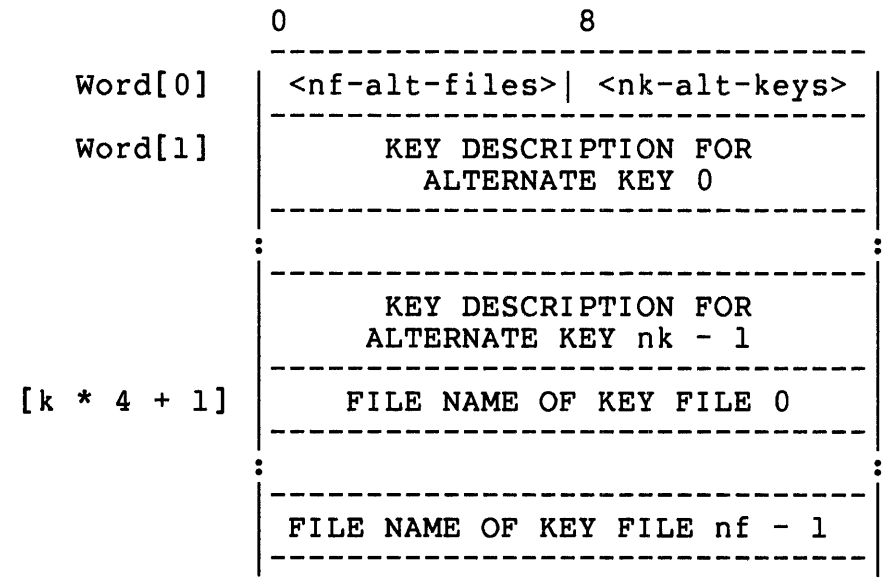
Considerations

- <key-sequenced-params> Array Format



<key-len> (INT:value)	is the length, in bytes, of the record's primary-key field. This length can be no larger than 255 bytes.
<key-offset> (INT:value)	is the number of bytes from the beginning of the record to where the primary-key field starts.
<index-block-len> (INT:value)	is the length, in bytes, of each index block in the file. <index-block length> must be a multiple of 512 and cannot be greater than 4096. If 0 is specified, then the value of <data-block length> is used as the <index-block length>.

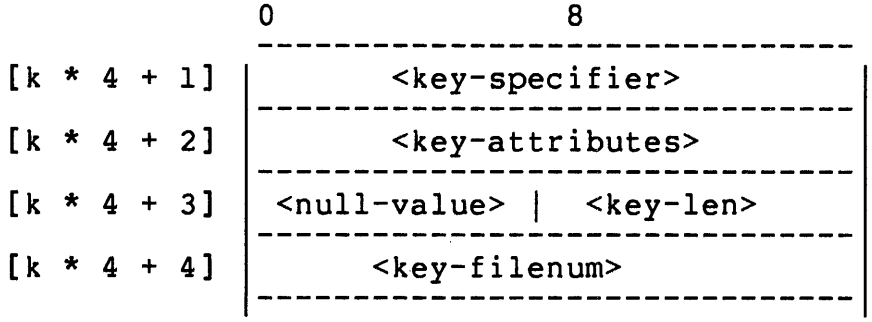
- <alternate-key-params> Array Format



<nf-alt-files> a one-byte value, specifies the number of alternate-key files for this primary file.

<nk-alt-keys> a one-byte value, specifies the number of alternate-key fields in this primary file.

The key description for key k consists of four words, each of the form:



<key-specifier> (INT:value) is a two-byte value that uniquely identifies this alternate-key field. This value is passed to the KEYPOSITION procedure for references to this key field.

<key-attributes> (INT:value) describes the key:

.<0> = 1 means a null value is specified. See "<null value>."

.<1> = 1 means the key is unique. If an attempt is made to insert a record that duplicates an existing value in this field, the insertion is rejected with a "duplicate record" error.

.<2> = 1 means ENSCRIBE cannot perform automatic updating of this key.

.<3> must be 0.

.<4:15> = <key-offset>. This specifies the number of bytes from the beginning of the record where this key field starts.

<null value> a one-byte value, is used to specify a null value if <key-attributes>.<0> = 1.

During a WRITE operation, if a null value is specified for an alternate-key field, and the null value is encountered in all bytes of this key field, the file system does not enter the reference to the record in the alternate-key file. (If the file is read via this alternate-key field, records containing a null value in this field will not be found.)

During a WRITEUPDATE operation (<write-count> = 0), if a null value is specified, and the null value is encountered in all bytes of this key field within <buffer>, the file system deletes the record from the primary file but does not delete the reference to the record in the alternate file.

<key-len> a one-byte value, specifies the length, in bytes, of this key field.

<key-filenum> (INT:value) is the relative number in the alternate-key parameter array of this key's alternate-key file. The first alternate-key file's <key-filenum> = 0.

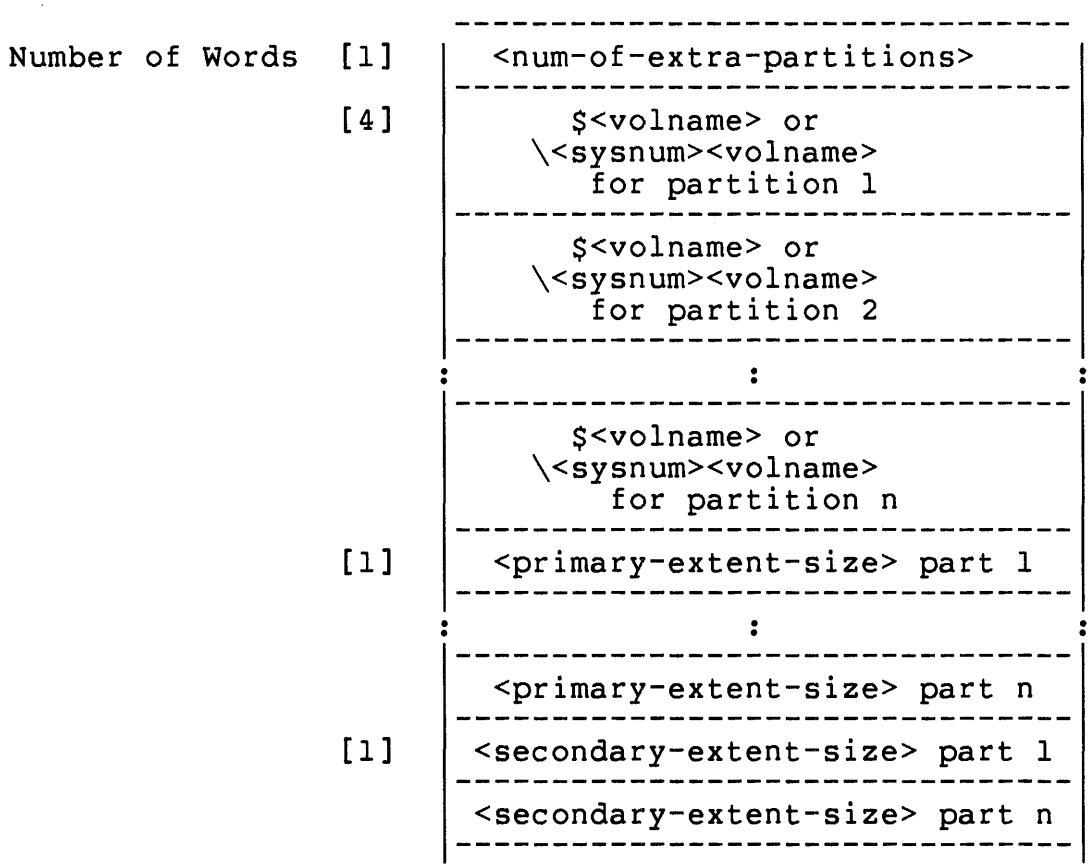
The file identifier for file f consists of 12 words, beginning at:

[nk * 4 + 1 + f * 12]

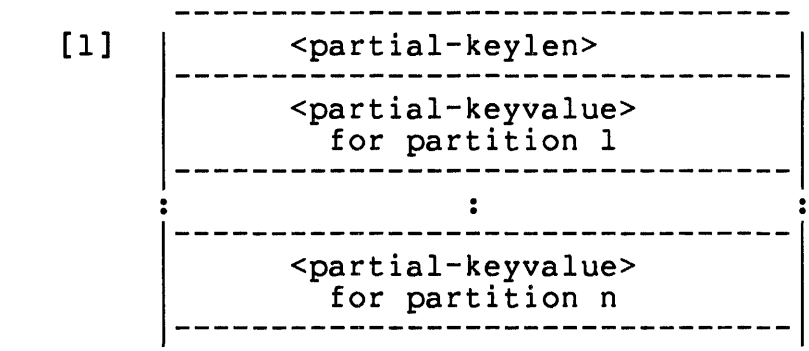
In this file identifier, the following is true:

```
<filename[0:3]>  = $<volname><blank-fill>
                              or
                   \<sysnum><volname><blank-fill>
<filename[4:7]>  =  <subvol-name><blank-fill>
<filename[8:11]> =  <disc-filename><blank-fill>
```

- <partition-params> Array Format



This sequence must be included in the partition-parameters array for key-sequenced files, but it can be omitted for other file types:



<num-of-extra-partitions> (INT:value) — is the number of extra volumes (other than the one specified in the <filename> parameter) on which the file resides. The maximum value permitted is 15. Note that every other parameter in the partition array (except <partial-keylen>) must be specified <num-of-extra-partitions> times.

$<volname>
or
\<sysnum><volname> — eight bytes blank-filled, is the name of the disc volume (including the dollar sign ($) or backslash (\)) where the particular partition is resides.

<primary-extent-size> (INT:value) — is the size of the primary extent for the particular partition.

<secondary-extent-size> (INT:value) — is the size of the secondary extents for the particular partition. Specifying 0 results in the <primary-extent-size> value being used.

The remaining parameters are required for key-sequenced files but can be omitted for all other file types:

<partial-keylen> (INT:value) — is the number of bytes of the primary key of a key-sequenced file that are used to determine which partition of the file contains a particular record. The minimum value for <partial-keylen> is 1.

<partial-key-value> (INT:value) — for <partial-key length> bytes, specifies the lowest key value that is allowed for a particular partition.

Each <partial-key-value> in <partition-parameters> must begin on a word boundary.

For an alternate-key-file, <partial-key-value> must begin with the <key-specifier> for the alternate key. For example, if <key-specifier> = AB, a partial-key value of 123 becomes a <partial-key-value> of AB123.

- File Pointer Action

  end-of-file pointer := 0D;

- Disc Allocation With CREATE

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

- CREATE Failure

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

- Altering File Security

  The file is created with the caller's process file security that can be examined and set with the PROCESSFILESECURITY procedure. Once a file has been created its file security can be altered by opening the file and issuing the appropriate SETMODE and SETMODENOWAIT functions.

- Minimum Extent Size

  If a file's index-block size is not the same as its data-block size, then no extent size should be smaller than the larger of the two block sizes. Otherwise, multiple extents are allocated every time a block is acquired, and some operations (such as the File Utility Program (FUP) LOAD command on a key-sequenced file) returns file system error 21 (illegal <count> specified).

- Odd Unstructured Files

  An odd unstructured file permits reading and writing of odd byte counts and positioning to an odd byte address.

  When creating unstructured files, the value passed for <file-type>.<12> determines how all subsequent reading, writing, and positioning operations to the file work.

  If <file-type>.<12> is passed as 1 and <file-type>.<13:15> is all zeros, an odd unstructured file is created.

  If <file-type>.<12> is passed as 1, the values of <record-specifier>, <read-count>, and <write-count> are all interpreted exactly; for example, a <write-count> or <read-count> of 7 transfers exactly 7 bytes.

- Even Unstructured Files

  A file must be positioned to an even byte address; otherwise, FILEINFO returns a file system error (bad address).

  If <file-type>.<13:15> is passed to CREATE and is all zeros (specifying an unstructured file), and <file-type>.<12> is 0, then an even unstructured file is created.

If <file-type>.<12> is passed as a 0, the values of <record-specifier>, <read-count>, and <write-count> are each rounded up to an even number before the operation begins; for example, a <write-count> or <read-count> of 7 is rounded up to 8, and 8 bytes are transferred.

If you use the FUP CREATE or COMINT CREATE command to create the file, it creates an even unstructured file.

Example

```
CALL CREATE ( DISC^FNAME , PRI^EXT , FILE^CODE , SEC^EXT
            , FILE^TYPE , REC^LEN , DATA^BLK^LEN , KEY^PARAMS );
```

Related Programming Manual

For programming information about the CREATE file system procedure, refer to the ENSCRIBE Programming Manual and the GUARDIAN Operating System Programmer's Guide.

CREATEPROCESSNAME PROCEDURE

The CREATEPROCESSNAME procedure returns a unique process name suitable for passing to the NEWPROCESS and NEWPROCESSNOWAIT procedures. This type of naming (as opposed to a predefined process name) is used when the name of a process pair does not need to be known to other processes in the system (for example, in a program run as several process pairs). This process name must be passed in the <name> parameter, not the <filename> parameter, of the NEWPROCESS procedure.

The syntax for CREATEPROCESSNAME is:

```
CALL CREATEPROCESSNAME ( <process-name> );              ! o

<process-name>                output

  INT:ref:3

  is an array where a system-generated process name returns.
  <process-name> is of the form:

    $Zddd

  where

    "d" represents an ASCII numeric character.

                          NOTE

      CREATEPROCESSNAME ensures that the next character
      position after the last "d" is a blank.
```

Condition Code Settings

< (CCL) indicates that the address passed for <process-name> is out of bounds, or the operating system could not access the the destination control table (DCT).

= (CCE) indicates the CREATEPROCESSNAME was successful.

> (CCG) does not return from CREATEPROCESSNAME.

Considerations

- Process Names and CREATEPROCESSNAME

  You use names created by CREATEPROCESSNAME when the process must be named, but the name of that process does not need to be predefined, that is, known by any other process or process pair.

  NOTE

  Calling CREATEPROCESSNAME does not enter the process name into the DCT.

- Creating Pseudo-Temporary Disc File Names

  The CREATEPROCESSNAME procedure is also useful for creating "pseudo-temporary" disc file names. You might use this type of naming when two processes want to use the same file, but each opens the file exclusively.

  If a standard temporary file name is used, the file is purged when the first process closes it because there are no other OPENs for the file. The second process is then unable to access the file; for example:

```
INT .TEMP^FNAME[0:11] := ["$VOL1 ", 9 * ["  "]];
      .
      .
      .
CALL CREATEPROCESSNAME ( TEMP^FNAME[4] );  ! returns $zddd.
TEMP^FNAME[4].<0:7> := "Z";                ! make zzddd subvol.
TEMP^FNAME[8] ':=' TEMP^FNAME[4] FOR 4;    ! make file name.
CALL CREATE ( TEMP^FNAME );
IF < THEN ... ; ! error.
      .
      .
      .
```

  The name returned from the above input is:

```
$VOL1  ZZddd    ZZddd
```

Example

See "Considerations."

Related Programming Manual

For programming information about the CREATEPROCESSNAME process control procedure, refer to the GUARDIAN Operating System Programmer's Guide.

CREATEREMOTENAME PROCEDURE

The CREATEREMOTENAME procedure supplies a process name that is unique for the specified system in a network. (This process name goes into the <name> parameter, not the <filename> parameter, of the NEWPROCESS procedure.)

The syntax for CREATEREMOTENAME is:

```
CALL CREATEREMOTENAME ( <name>                          ! o
                       ,<sysnum> );                     ! i
```

<name> output

INT:ref:3

is an array where CREATEREMOTENAME returns a system-generated process name (in local form) that is unique for the designated system. <name> is of the form:

$Zddd

where

"d" represents an ASCII numeric character.

NOTE

CREATEREMOTENAME ensures that the next character position after the last "d" is a blank.

<sysnum> input

INT:value

is a value that specifies the system number for which the process name is to be created.

Condition Code Settings

< (CCL) indicates that the remote the destination control table (DCT) could not be accessed.

= (CCE) indicates that CREATEREMOTENAME was successful.

> (CCG) does not return from CREATEREMOTENAME.

Considerations

- Remote Process Name Characteristics

  CREATEREMOTENAME creates a process name in local form. This name can be passed directly to the NEWPROCESS procedure as the <name> parameter in order to create a remote process having that name. It is unnecessary to append a system name to the process name since the physical location of the program file specified in the NEWPROCESS <filename> includes the system number.

- Remote System DCT

  The creation of a process name does not make an entry in the remote system's DCT.

Example

```
CALL CREATEREMOTENAME ( NAME , SYS^NUM );
```

Related Programming Manual

None

CREATORACCESSID PROCEDURE

The CREATORACCESSID procedure is used to obtain the accessor ID of the process that created the calling process.

The syntax for CREATORACCESSID is:

```
<accessor-id> := CREATORACCESSID;


<accessor-id>                 returned value

  INT

  returns the accessor ID of the caller's creator in the
  following form:

    <accessor-id>.<0:7>  = group number
                 .<8:15> = user number
```

Condition Code Settings

The condition code has no meaning following a call to CREATORACCESSID.

Considerations

- Accessor ID Returned From CREATORACCESSID and MOM

  The accessor ID returned from CREATORACCESSID is that of the calling process's actual creator, which is not necessarily the same as that returned from a call to the MOM procedure.

Example

```
CREATOR^ID := CREATORACCESSID;
```

Related Programming Manual

For programming information about the CREATORACCESSID security procedure, refer to the GUARDIAN Operating System Programmer's Guide.

CURRENTSPACE PROCEDURE

The CURRENTSPACE procedure returns a stack-marker ENV register and a string (in ASCII) containing the space ID of the caller.

The syntax for CURRENTSPACE is:

```
{ <stack-env> := } CURRENTSPACE [ ( <ascii-space-id> ); ]    ! o
{ CALL          }
```

<stack-env> returned value

INT

is the calling procedure's space ID in the stack marker ENV register format.

```
ENV.<4>          ! library bit.
ENV.<7>          ! system code bit.
ENV.<11:15>      ! space ID bits.
```

For more information about space identifiers and the details of these bits, refer to the System Description Manual.

<ascii-space-id> output

STRING:ref:5

is an ASCII string in the form:

<map>.<#>

where

<map> is one the following:

UC	indicates user code.
UL	indicates user library.
SC	indicates system code.
SL	indicates system library.

<#> is the octal space number number in ASCII; for example:

UC.01 or SL.33

Condition Code Settings

The condition code has no meaning following a call to CURRENTSPACE.

Example

```
MY^SPACE := CURRENTSPACE;
```

Related Programming Manual

For information about the CURRENTSPACE procedure, refer to the System Description Manual.

DEALLOCATESEGMENT PROCEDURE

The DEALLOCATESEGMENT procedure deallocates an extended data segment when it is no longer needed by the calling process.

The syntax for DEALLOCATESEGMENT is:

```
CALL DEALLOCATESEGMENT ( <segment-id>                ! i
                       ,[ <flags> ] );               ! o
```

<segment-id> input

INT:value

is the segment number of the segment, as specified in the call to ALLOCATESEGMENT that created it.

<flags> input

INT:value

if present, has the form:

```
  <0:14>   = must be 0.
    <15>   = 1  indicates that dirty pages in memory are not to
                be copied to the swap file (see ALLOCATESEGMENT
                procedure).
           = 0  indicates that dirty pages in memory are to be
                copied to the swap file.
```

If omitted, this parameter defaults to 0.

Condition Code Settings

The condition code has no meaning following a call to DEALLOCATESEGMENT.

Considerations

- <flags> Parameter

  The <flags>.<15> = 1 option is used to improve performance when the swap file is a permanent file or a temporary file that is opened concurrently by an application. Following the DEALLOCATESEGMENT call, the contents of the swap file are unpredictable. If the DEALLOCATESEGMENT call causes a purge of a temporary file, the GUARDIAN operating system does not write the dirty pages (that is, pages that are being used) out to the file.

- Before deallocating a segment, this procedure removes all memory access breakpoints set in that segment.

- Segment Deallocation

  When a segment is deallocated, the swap file end of file (EOF) is set to the larger of (1) the EOF when the file is opened by ALLOCATESEGMENT or (2) the end of the highest numbered page that is written to the swap file. All file extents beyond the EOF that did not exist when the file was opened are deallocated.

Example

```
CALL DEALLOCATESEGMENT ( SEGMENT^ID );
```

SEGMENT^ID refers to the segment number specified in the call to ALLOCATESEGMENT.

Related Programming Manual

For programming information about the DEALLOCATESEGMENT memory management procedure, refer to the GUARDIAN Operating System Programmer's Guide.

DEBUG PROCEDURE

The debug facility can be invoked directly by calling the DEBUG procedure.

The syntax for DEBUG is:

```
CALL DEBUG;
```

NOTE

The GUARDIAN operating system provides a debugging facility that responds to debug events by passing control to one of two debugging utilities: DEBUG or INSPECT. DEBUG is a low-level debugger. INSPECT is an interactive symbolic debugger that lets you control program execution, display values, and modify values in terms of source-language symbols.

DEBUG

While a process is in the debug state, you can interactively display and modify the contents of the process's registers (the process's data area and set other breakpoints. To debug a program, the user must have EXECUTE access to run the program and READ access to the program object file.

There are five ways to force a process into the debug state:

- Run a program through the command interpreter using the RUND (RUN DEBUG) command. The process enters the debug state before the first instruction of the MAIN procedure executes.
- Run a program through the NEWPROCESS procedure, and set <priority>.<0>, the debug bit, to 1. The process enters the debug state before the first instruction of the MAIN procedure executes.
- Run a program through the command interpreter. While the program is executing, press the BREAK key. The command interpreter returns to the command input mode. Find the <cpu,pin> of the process, and type in DEBUG <cpu,pin>.
- In the source program, write an explicit call to the DEBUG procedure.
- When a process is in the debug state, specify a breakpoint. When that breakpoint is hit, the process enters the DEBUG state.

For a description of the debug facility and instructions for using it, see the DEBUG Manual for your system.

INSPECT

You can use INSPECT by setting the INSPECT attribute associated with a process. The value of a process's INSPECT attribute can be set with:

- The ?INSPECT or ?SAVEABEND compiler directive
- The BINDER SET INSPECT or SET SAVEABEND commands during a binding session
- The COMINT SET INSPECT command before the RUN command that starts the process
- The INSPECT parameter of the RUN command that starts the process
- A new parameter in the NEWPROCESS procedure that allows you to specify INSPECT as the default debugger

Processes inherit the INSPECT attribute from their ancestor processes. For a description of the INSPECT facility and instructions for its use, see the INSPECT Interactive Symbolic Debugger User's Guide.

Example

```
CALL DEBUG;
```

Related Programming Manual

For information about the DEBUG vacility, refer to the DEBUG Manual For information about the INSPECT facility, refer to the INSPECT Interactive Symbolic Debugger User's Guide.

DEBUGPROCESS PROCEDURE

The DEBUGPROCESS procedure is used to invoke the debug facility on a process.

The syntax for DEBUGPROCESS is:

```
CALL DEBUGPROCESS ( <process-id>                    ! i
                  ,<error>                          ! o
                  ,[ <term> ]                       ! i
                  ,[ <now> ]  );                    ! i
```

<process-id> input

INT:ref:4

is the process ID of the process to be debugged. It can be in a timestamp or named and in local or remote format.

<error> output

INT:ref:1

returns a file system error number indicating the outcome of the process debug attempt.

<term> input

INT:ref:12

is the name of the debug home terminal. If omitted, the caller's home terminal is used.

<now> parameter--Super Group ID Only

<now>	input
INT:value	
The caller's process access ID must be super ID (255, 255) to use this parameter.	
If you supply 1, the process should be debugged now. If omitted, the normal debug sequence is executed.	

Condition Code Settings

The condition code has no meaning following a call to DEBUGPROCESS (see the <error> parameter description).

Example

```
CALL DEBUGPROCESS ( PROC^ID , ERROR^NUM , MYTERM , ONE );
```

Related Programming Manual

None

DEFINELIST PROCEDURE

The DEFINELIST procedure is used only when the application process is acting as a supervisor or tributary station in a centralized multipoint configuration.

- Within a supervisor station, DEFINELIST specifies the station addresses of each tributary station that the application process wishes to communicate with.
- Within a tributary station, DEFINELIST specifies the station addresses that the particular line responds to.

The addresses are in the form of a "station list" array whose name passes to the DEFINELIST procedure by way of the DEFINELIST calling sequence.

The syntax for DEFINELIST is:

```
CALL DEFINELIST ( <filenum>                                  ! i
                 ,<address-list>                             ! i
                 ,<address-size>                             ! i
                 ,<num-entries>                              ! i
                 ,<polling-count>                            ! i
                 ,<polling-type> );                          ! i
```

<filenum> input

INT:value

is the name of the one-word integer variable specified in the OPEN call that opened the line.

<address-list> input

INT:ref:*

is the name of an integer array containing either:

- Polling addresses and selection addresses (refer to the ENVOY Byte-Oriented Protocols Reference Manual for a description of this array)
- one or more station addresses (refer to the ENVOYACP Bit-Oriented Protocols Reference Manual for a description of this array)

→

<address-size> input

INT:value

specifies the size, in words, of an entry in the <station-list> array. Note that the entry size varies somewhat from one protocol to another.

<num-entries> input

INT:value

specifies the total number of entries in the <station-list> array.

<polling-count> input

INT:value

specifies the number of polling addresses in the <station-list> array. This parameter has no meaning when used for ENVOYACP bit-oriented protocols.

<polling-type> input

INT:value

For a supervisor station, specifies the number of times that the tributary stations with polling addresses in the <station-list> array are to be polled when the line is in the control state, and the supervisor station issues a READ call:

```
      0 = poll continuously
  1-127 = number of polling cycles.
```

For tributary stations, this parameter has no functional effect; a dummy argument must still be supplied, however, for each station except ENVOY's multipoint tributary. In this case, the <polling-type> can be:

```
  0 = RVI (reverse interrupt)
  1 = WACK (wait for acknowledgment)
  2 = NAK (negative acknowledge)
```

Condition Code Settings

< (CCL) indicates that an error occurred (call FILEINFO).

= (CCE) indicates that the DEFINELIST procedure was executed successfully.

> (CCG) does not return from DEFINELIST.

Considerations

- Call DEFINELIST after the call to OPEN but before the first call to READ or WRITE.

Examples

```
CALL DEFINELIST ( FNUM , ADDR^LIST , ADDR^SIZE , NUM^ENTRIES
                , POLLING^COUNT , POLLING^TYPE );
```

Related Programming Manuals

For programming information about the DEFINELIST system procedure, refer to the data-communication manuals.

DEFINEPOOL PROCEDURE

The DEFINEPOOL procedure designates a portion of a user's stack or an extended data segment for use as a pool.

The syntax for DEFINEPOOL is:

```
<status> := DEFINEPOOL ( <pool-head>              ! o
                        ,<pool>                   ! i
                        ,<pool-size> );           ! i
```

<status> returned value

INT

returns a status word having one of the following values:

0 = no error
1 = bounds error on <pool-head>
2 = bounds error on <pool>
3 = invalid <pool-size>
4 = <pool-head> and <pool> overlap.

<pool-head> output

INT .EXT:ref:19

is a 19-word array to be used as the pool header; GETPOOL and PUTPOOL use this array to manage the pool.

<pool> input

INT .EXT:ref:*

is the first word of the memory space to be used as the pool.

<pool-size> input

INT(32):value

is the size of the pool in bytes. This number must be a multiple of 4 bytes and cannot be less than 32 or greater than %10000000D.

Condition Code Settings

The condition code setting has no meaning following a call to DEFINEPOOL. (See the <status> parameter definition.)

Considerations

- Stack Addresses Converted to Extended Addresses

  If <pool-head> or <pool> is in the user data stack, the Tandem Application Language compiler automatically converts data stack addresses to extended addresses.

- Dynamic Memory Allocation

  Several GUARDIAN procedures 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, the <pool-header>, that 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 can 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.

  The programmer must be careful to use only the currently reserved blocks of the pool, or the pool structure is corrupted and unpredictable results occur. If multiple pools are defined, do not return reserved blocks to the wrong pool. For debugging purposes, a special call to GETPOOL 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 determining the proper size of a pool. Application programs should not depend on the pool data structures, since they are subject to change. The program should use only the procedural interfaces described on the following pages.

  The requested block size is rounded up to a multiple of 4 bytes, at a minimum of 28 bytes. This reduces pool fragmentation, but when the program is allocating small blocks, it can waste memory space.

One extra word is allocated for a boundary tag at the beginning and end of each block; thus, the minimum pool block size is 32 bytes. This tag serves three purposes:

1. It contains the size of each block so that the program does not need to specify the length of the block when releasing it.

2. It serves as a check to ensure that the program does not erroneously use more memory than the block contains (although it does not stop the program from overwriting).

3. It provides for efficient coalescing of adjacent free blocks.

In GETPOOL, the free block list is searched for the first block sufficiently large enough 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 used for the request.

In summary, the pool space overhead on each block can be substantial if very small blocks are allocated. An exact formula is:

```
ALLOCATED := ($MAX (REQUEST + 7, 32) /4) * 4
```

where REQUEST is the original request size in bytes; the allocated blocks are also measured in bytes.

Although they can also be used to manage the allocation of a collection of equal-sized blocks, these procedures are not recommended for that purpose, because they can consume more processor time and pool memory than user-written routines designed for that specific task.

Example

```
STATUS := DEFINEPOOL ( POOL^HEAD , POOL , 2048D );
```

Related Programming Manual

For programming information about the DEFINEPOOL memory management procedure, refer to the GUARDIAN Operating System Programmer's Guide.

DELAY PROCEDURE

The DELAY procedure permits a process to have itself suspended for a timed interval.

The syntax for DELAY is:

```
CALL DELAY ( <time-period> );                    ! i

<time-period>                input

  INT(32):value

  specifies the time period, in .01-second units, for which the
  the caller of DELAY is to be suspended.
```

Condition Code Settings

The condition code has no meaning following a call to DELAY.

Considerations

- <time-period> Value of <> 0D

  A value of less than or equal to 0D results in no delay as such but changes this process's execution state from active to ready in order to permit other processes of the same priority to run.

Example

```
CALL DELAY ( 1000D );                ! suspend for 10 seconds.
```

Related Programming Manual

None

DEVICEINFO PROCEDURE

The DEVICEINFO procedure is used to obtain the device type and the physical record length of a file. The file can be opened or closed.

The syntax for DEVICEINFO is:

```
CALL DEVICEINFO ( <filename>                    ! i
                 ,<devtype>                     ! o
                 ,<physical-recordlen> );       ! o
```

<filename> input

INT:ref:8

is an array containing the name of the device whose characteristics are to be returned. Any form of internal file name is permitted. For disc files, only the first eight characters (that is, the volume name) are significant; however, the remaining eight characters still must be in a valid file name format. If a logical device number is specified, the last eight characters must be blanks.

<devtype> output

INT:ref:1

returns the device type of the associated file in this form:

```
      .<0> = demountable
      .<1> = audited disc
    .<4:9> = device type
  .<10:15> = device subtype
```

Refer to Appendix B for a list of the device types.

<physical-recordlen> output

INT:ref:1

returns the physical record length associated with the file. If the physical record length is for:

nondisc devices	<physical-recordlen> is the configured record length.

→

disc files	<physical-recordlen> is the maximum possible transfer length. The transfer length is equal to the configured buffer size for the device (either 2048 or 4096 bytes). (For an ENSCRIBE disc file, the logical record length can be obtained through the FILERECINFO procedure.)
processes and $RECEIVE file	a length of 132 is returned in <physical-recordlen>. This is the system convention for interprocess files.

Condition Code Settings

The condition code has no meaning following a call to DEVICEINFO.

Example

```
CALL DEVICEINFO ( INFILE , DEVTYPE , RECLENGTH );
```

Related Programming Manuals

For programming information about the DEVICEINFO file system procedure, refer to the GUARDIAN Operating System Programmer's Guide and the ENSCRIBE Programming Manual.

DEVICEINFO2 PROCEDURE

The DEVICEINFO2 procedure is used to obtain the device type and the physical record length of a file (the file can be opened or closed) and to determine whether the volume is formatted for the DP1 or the DP2 disc process.

The syntax for DEVICEINFO2 is:

```
CALL DEVICEINFO2 ( <filename>                    ! i
                 ,<devtype>                      ! o
                 ,<physical-recordlen>           ! o
                 ,<discprocess-version> );       ! o
```

<filename> input

INT:ref:8

is an array containing the name of the device whose characteristics are to be returned. Any form of internal file name is permitted. For disc files, only the first eight characters (that is, the volume name) are significant; however, the remaining eight characters must still be in a valid file name format. If a logical device number is specified, the last eight characters must be blanks.

<devtype> output

INT:ref:1

returns the device type of the associated file in this form:

```
      .<0> = demountable
      .<1> = audited disc
    .<4:9> = device type
  .<10:15> = device subtype
```

Refer to Appendix B for a list of the device types.

<physical-recordlen> output

INT:ref:1

returns the physical record length associated with the file.

→

```
  If the physical record length is for:

  nondisc          <physical-recordlen> is the configured
  devices          record length.

  disc files       <physical-recordlen> is the maximum possible
                   transfer length.  The transfer length is equal
                   to the configured buffer size for the device
                   (either 2048 or 4096 bytes).  (For an ENSCRIBE
                   disc file, the logical record length can be
                   obtained through the FILERECINFO procedure.)

  processes        a length of 132 is returned in
  and $RECEIVE     <physical-recordlen>.  This is the system
  file             convention for interprocess files.

<discprocess-version>        output

  INT:ref:1

  returns the disc process version for disc devices
  (<device-type>.<4:9> = 3).

    <discprocess-version> = 0  DP1 disc process
                          = 1  DP2 disc process
```

Condition Code Settings

The condition code has no meaning following a call to DEVICEINFO2.

Example

```
CALL DEVICEINFO2 ( INFILE , DEVTYPE , RECLENGTH , D^VERSION );
```

Related Programming Manual

None

EDITREAD PROCEDURE

The EDITREAD procedure reads text lines from an EDIT file (file type = 101).

Text lines are transferred, in ascending order, from the text file to a buffer in the application program's data area. One line is transferred by each call to EDITREAD. EDITREAD also returns the sequence number associated with the text line and performs checks to ensure that the text file is valid.

The EDIT file can be opened nowait. However, a call to EDITREAD completes before returning to the application program; it is not completed with a call to AWAITIO.

NOTE

Before EDITREAD is called, a call to EDITREADINIT must complete successfully.

The syntax for EDITREAD is:

```
<status> := EDITREAD ( <edit-controlblk>        ! i
                      ,<buffer>                 ! o
                      ,<bufferlen>              ! i
                      ,<sequence-num> );        ! o
```

<status> returned value

INT

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

>= 0 indicates that the reading of the file was successful. This is the actual number of characters in the text line. However, no more than <bufferlen> bytes are transferred into <buffer>.

< 0 indicates an unrecoverable error, where:

-1 = end of file encountered
-2 = error occurred while reading
-3 = text file format error

→

-4 = sequence error. The sequence number of the line just read is less than its predecessor.
-5 = checksum error. EDITREADINIT was not called or the user has altered the edit control block. (This will not be returned if processing a reposition.)

<edit-controlblk> input

INT:ref:*

is an uninitialized array that is declared globally. The length in words, of the edit control block must be at least 40 plus <bufferlen> divided by 2.

<buffer> output

STRING:ref:*

is an array where the text line is to be transferred.

<bufferlen> input

INT:value

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

<sequence-num> output

INT(32):ref:1

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

Condition Code Settings

The condition code has no meaning following a call to EDITREAD.

Example

```
COUNT := EDITREAD ( CONTROL^BLOCK , LINE , LENGTH , SEQ^NUM );
```

If reading the file is successful, a count of the number of bytes in the text line returns in COUNT, the text line returns in the array LINE, and the sequence number returns in SEQ^NUM.

Related Programming Manual

None

EDITREADINIT PROCEDURE

The EDITREADINIT procedure is called to prepare a buffer in the application program's data area for subsequent calls to EDITREAD.

The application program designates an array to be used as an edit control block. The edit control block is used by the EDITREAD procedure for storing control information and as an internal buffer area.

The EDIT file can be opened nowait. However, a subsequent call to EDITREADINIT completes before returning to the application; it is not completed with a call to AWAITIO.

The syntax for EDITREADINIT is:

```
<status> := EDITREADINIT ( <edit-controlblk>        ! i
                         ,<filenum>                 ! i
                         ,<bufferlen> );            ! i
```

<status> returned value

INT

is a value indicating the outcome of EDITREADINIT. Values returned into <status> are:

```
   0 = successful (OK to read)
  -1 = end of file detected (empty file)
  -2 = I/O error
  -3 = format error (not EDIT file)
  -4 = sequence number error
```

<edit-controlblk> input

INT:ref:*

is an uninitialized array that is declared globally. Forty words of the edit control block are used for control information. The remainder is used as an internal buffer by EDITREAD. The length, in words, of the edit control block must be at least 40 plus <bufferlen> divided by 2. This is the same array as specified in the <edit controlblk> parameter to EDITREAD.

→

<filenum> input

INT:value

is the number of an open file that identifies the text file to be read.

<bufferlen> input

INT:value

is the size, in bytes, of the internal buffer area used by EDITREAD. This parameter determines the amount of data that EDITREAD reads from the text file on disc (not the amount of data transferred into the buffer specified as a parameter to EDITREAD). The size of the internal buffer area must be a power of two, from 64 to 2048 bytes (that is, 64, 128, 256, ..., 2048).

Condition Code Settings

The condition code has no meaning following a call to EDITREADINIT.

Example

```
STAT := EDITREADINIT ( CONT^BLOCK , FNUM , BUF^LEN );
```

Related Programming Manual

None

ENDTRANSACTION PROCEDURE

ENDTRANSACTION commits the data base changes associated with a transaction identifier. When this procedure is called by the process (or its backup) that issued BEGINTRANSACTION, the Transaction Monitoring Facility (TMF) attempts to commit the transaction. If the action completes successfully, the changes made by the transaction are permanent, and the locks held for the transaction are released. (Locks are held until ENDTRANSACTION returns.)

The syntax for ENDTRANSACTION is:

```
<status> := ENDTRANSACTION;
```

<status>	returned value
INT	returns a 0, if the transaction ended successfully, or a file system error number. (See the System Messages Manual for a a list of all file system errors.)

Condition Code Settings

The condition code has no meaning following a call to ENDTRANSACTION.

Considerations

- Waited Operation

  ENDTRANSACTION is, by default, a waited operation unless the calling process has the transaction pseudofile (TFILE) open at the time of the ENDTRANSACTION call. ENDTRANSACTION waits until the record is written to the audit file and TMF can ensure the transaction will commit. However, it does not wait for locked records to be unlocked.

- Obtaining a Link Control Block (LCB)

  If the procedure fails to obtain main-memory space for an LCB, the call fails and returns file system error 30.

- Requesting Process and Current-Transaction Identifier

  If the requesting process has no current-transaction identifier, the call fails and returns file system error 75.

- Invalid or Obsolete Transaction Identifier

  If the transaction was not begun by the process that issued ENDTRANSACTION (or its backup) or the transaction is no longer in the system, this call returns file system error 78.

- Nowait Operation

  If an outstanding (that is, incomplete) operation is pending against a process file or disc for this transaction, the call fails and returns file system error 81.

- Transaction Aborts

  If the parent process (BEGINTRANSACTION process) of this transaction fails, the call fails and returns with file system error 90.

- Path in Network Node Is Down

  If the path to a participating network node is down, the transaction aborts, and the call returns file system error 92.

- Spanning Too Many Audit-Trail Files

  If the transaction spans too many audit-trail files, the transaction is aborted and the call returns file system error 93.

- Operator Command Response

  If an operator aborts the transaction through the TMFCOM "ABORT TRANSACTION" command, the system aborts the transaction, and the call returns file system error 94.

- Calling ABORTTRANSACTION Before ENDTRANSACTION

  If a previous call to ABORTTRANSACTION is made before the call to ENDTRANSACTION, the call aborts and returns file system error 97.

Example

```
STATUS := ENDTRANSACTION;
```

Related Programming Manual

For information about the ENDTRANSACTION procedure, refer to the Transaction Monitoring Facility (TMF) Reference Manual.

ENFORMFINISH PROCEDURE

ENFORMFINISH is called once to terminate the interface to ENFORM.

The syntax for ENFORMFINISH is:

```
CALL ENFORMFINISH ( <ctlblock> );                    ! i

<ctlblock>                    input

  INT:ref:18

  is the same 18-word integer array control block supplied to
  ENFORMSTART for global storage among all ENFORM procedure
  calls.  The host language program must not change the control
  block between calls to ENFORM.
```

Condition Code Settings

The condition code has no meaning following a call to ENFORMFINISH.

Example

```
CALL ENFORMFINISH ( CNTL^BLOCK );     !control block
```

Related Programming Manual

For programming information about the ENFORMFINISH procedure, refer to the ENFORM User's Guide.

ENFORMRECEIVE PROCEDURE

The ENFORMRECEIVE procedure provides records to the host application program one at a time.

The syntax for ENFORMRECEIVE is:

```
{ <count> := } ENFORMRECEIVE ( <ctlblock>              ! i
{ CALL       }                ,<buffer> );             ! i, o
```

<count> returned value

INT

returns a byte count for the length of the record retrieved (all FIND output records are the same length). A zero value means that all records have been returned.

<ctlblock> input

INT:ref:18

is the same 18-word integer array control block supplied to ENFORMSTART for global storage among all ENFORM procedure calls. The host application program must not change the control block between calls to ENFORM.

<buffer> input, output

INT:ref:*

is a pointer to the receiving buffer in the host application program for an output record. It must be at least as long as the value given for <buffer-length> in ENFORMSTART. The parameter <buffer> should be declared as a type of FORTRAN record if you want to access character information in a FORTRAN program.

Condition Code Settings

< (CCL) indicates a query processor error occurred; and <error-number> (passed to ENFORMSTART) contains an error number greater than 0 (see Appendix E).

= (CCE) indicates successful receipt of a target record.

> (CCG) indicates that no more target records exist.

Considerations

- Calls to ENFORMRECEIVE

  ENFORMRECEIVE sends the records to the host language program one at a time. Hence, ENFORMRECEIVE is repeatedly called until no more records exist, ENFORMFINISH is called, or an error condition occurs.

- Action When an ENFORM Error Occurs During Execution

  If an error occurs during the execution of the ENFORMRECEIVE procedure, the number of the error returns to the <error-number> supplied to the ENFORMSTART procedure. Any error conditions terminate the ENFORM program. If the query processor is dedicated, it is deleted. Refer to Appendix K for a list of possible ENFORM errors returned for the ENFORMRECEIVE procedure.

Example

```
COUNT := ENFORMRECEIVE ( CNTL^BLK , ORDER^PROCESS^REC );
```

Related Programming Manual

For programming information about the ENFORMRECEIVE procedure, refer to the ENFORM User's Guide.

ENFORMSTART PROCEDURE

The ENFORMSTART procedure initiates the interface of a host program with ENFORM.

The syntax for ENFORMSTART is:

```
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
```

<ctlblock> output

INT:ref:18

is an 18-word integer array control block that must be supplied for global storage across the ENFORM procedure calls. This same storage is used in ENFORMRECEIVE and ENFORMFINISH and any subsequent calls to ENFORMSTART. The host application program must not change this control block between calls to ENFORM.

<compiled-physical-filename> input

INT:ref:12

is a 12-word array that specifies the physical file containing the compiled query. The file name must be specified as 24 characters in length with the:

```
$volume name    = 8 characters (blank-filled if necessary)
subvolume name  = 8 characters (blank-filled if necessary)
disc file name  = 8 characters (blank-filled if necessary)
```

Refer to the Guardian Operating System Programmer's Guide for the exact form of an internal file name.

→

<buffer-length> input

INT:value

is the length, in bytes, of the buffer that the process uses to receive records through ENFORMRECEIVE. <Buffer-length> must be at least 6. If information in addition to the error number is desired, <buffer-length> must be at least 30. Refer to the ENFORM User's Guide for more information.

<error-number> output

INT:ref:1

is assigned an error number if an error condition occurs in ENFORMSTART or ENFORMRECEIVE. It is initialized to zero. <Error-number> should be declared globally to the ENFORM procedures so that it can be checked after the ENFORMSTART or ENFORMRECEIVE procedures return. The error messages are described in Appendix E.

<restart-flag> input

INT:value

is used when there is more than one query to be run by the host application program. A nonzero value causes the existing query processor to begin the next query with a new parameter and assign list, which is more economical than creating a new query processor for each query run. When the existing query processor is used to begin the next query, the parameter values for <ctlblock>, <qp-name>, <cpu>, and <priority> for the ENFORMSTART procedure must be identical to those for the initial call to the ENFORMSTART procedure, or they must not be used.

A zero value for <restart-flag> causes a new query processor to be created for each querry.

<param-list> input

INT:ref*

is a pointer to the parameter (name:value) list in a form equivalent to the PARAM message generated by the command interpreter. See the GUARDIAN Operating System Programming Guide for the correct format.

→

<assign-list> input

INT:ref:*

is a pointer to a sequence of one or more ASSIGN messages, each formatted as an ASSIGN message by the command interpreter. See the GUARDIAN Operating System Utilities Reference Manual for the correct format. These messages are preceded by a one-word header that contains a number equal to the total messages (31 or fewer) in the list.

<qp-name> input

INT:ref:4

is a four-word array that, if present, specifies the process name of a server query processor to use. If a query processor by that name does not exist, or if it cannot accept the query because the query processor is busy or the query exceeded its processing limits, an error results. If this parameter is omitted, a dedicated query processor for the query is created by ENFORMSTART and deleted by ENFORMFINISH. Refer to the ENFORM Reference Manual for information about a server query processor.

<cpu> input

INT:value

selects the CPU in which to run a dedicated query processor. If omitted, the query processor runs in the same CPU as the host application program. When <qp-name> is given, this parameter is ignored.

<priority> input

INT:value

assigns a priority for a dedicated query processor. The default is the priority assigned the host application program. When <qp-name> is given, this parameter is ignored.

<timeout> input

INT(32):value

if present, indicates the maximum time (in .01-second units) that the application process is willing to wait (that is, be suspended) for the query processor to begin processing the query.

If <timeout> is omitted, then the application process waits indefinitely for the query processor to begin the query.

<reserved-for-expansion> input

is reserved for expansion.

Condition Code Settings

< (CCL) means there is a problem. An error number representing the reason is found in <error-number>.

= (CCE) means the query processor successfully initialized.

> (CCG) is not used by ENFORMSTART.

Considerations

- <assign-list>--Overriding Names for Files to the Query

  <assign-list> is used to override the physical file name or file names for input files to the query. Any physical file name that is not expanded in the ASSIGN message uses the default volume and subvolume. If that is undesirable, the host language program must fill in the correct volume or subvolume after receiving ASSIGN messages from the command interpreter and before passing <assign-list> to ENFORMSTART. When <assign-list> is not used, the Tandem physical file name compiled into the query is used.

- <assign-list>--Generic File Names

  <assign-list> can contain generic file names. Refer to the ENFORM Reference Manual for information about generic files. If the generic file QUERY-QPSTATUS-MESSAGE is specified from a host language program, ENFORM uses the message text from $SYSTEM.SYSTEM.ENFORMMK. There is no way to specify a new message table from a host language program.

The physical file name and its exclusion specification (shared, protected, or exclusive) can be overridden. When not overridden, the query processor always opens the physical input files with shared mode.

- When is <timeout> Most Meaningful

  <timeout> is most meaningful to use in the case of a request to a server query processor named with <process-name> that might be busy processing another query at the time ENFORMSTART is called. If ENFORMSTART returns error 22, indicating that timeout occurred, the application process could revert to a dedicated query processor by again calling ENFORMSTART without the <process-name> parameter.

- Action When an Error Occurs During Execution

  If an error occurs during the execution of the ENFORMSTART procedure, the number of the error returns to <error-number>. Any error conditions terminate the query processor. If the query processor is dedicated, it is deleted. Refer to Appendix E for a list of possible ENFORM errors returned for ENFORMSTART.

Example

```
CALL ENFORMSTART ( CNT^BLK
                 , QUERY^FILENAME
                 , $LEN (ORDER^PROCESS^REC)
                 , ERROR
                 ,
                 , PARAM^LIST );
```

Related Programming Manual

For programming information about the ENFORMSTART procedure, refer to the ENFORM User's Guide.

EXPAND^SCREEN PROCEDURE

For users of the ENTRY or ENTRY520 screen formatter, the EXPAND^SCREEN procedure places the control sequences and variable data you want to display or default values, and optionally the entire form, into the application program's I/O buffer. The system I/O procedures perform the actual I/O.

The syntax for EXPAND^SCREEN is:

```
<num-bytes> := EXPAND^SCREEN ( @<screen-name>            ! i
                             , SCREEN                    ! o
                             , <buffer>                  ! o
                             , <rewrite-form> );         ! i
```

<num-bytes> returned value

INT

returns the number of bytes that are placed in the program's I/O buffer.

<screen-name> input

INT:value

is the address of the READ-only array that has the screen definition (refer to the ENTRY Screen Formatter Operating and Programming Manual or the ENTRY520 Screen Formatter Operating and Programming Manual for an explanation of "screen definition").

SCREEN output

STRING:ref:*

is the required array named SCREEN. If you omit this parameter, the screen displays the default values. If you give this parameter, the screen displays only the data entries moved into the SCREEN array. Data entries shorter than a defined field should be null-terminated.

→

<buffer>	output

STRING:ref:*

is the program's I/O buffer. Form control and data characters move here prior to terminal display.

<rewrite-form>	input

INT:value

is nonzero if both the protected form and the data-entry fields are written to the terminal. A zero means only to write the entry fields. For multiple inputs or outputs using the same form, it is faster to just rewrite the data-entry fields (<rewrite-form> = 0).

Condition Code Settings

The condition code has no meaning following a call to EXPAND^SCREEN.

Considerations

- Forms with no data-entry fields display at full brightness. Normally, the form is protected and is displayed "half bright," while the entry fields display at full intensity.

Example

```
NUM^BYTES := EXPAND^SCREEN ( @X , SCREEN , BUF , 1 );
```

Related Programming Manuals

For programming information about the EXPAND^SCREEN entry procedure, refer to the ENTRY Screen Formatter Operating and Programming Manual or the ENTRY520 Screen Formatter Operating and Programming Manual.

FILEERROR PROCEDURE

The FILEERROR procedure is used to determine if an I/O operation that completed with an error should be retried.

The syntax for FILEERROR is:

```
<status> := FILEERROR ( <filenum> );              ! i

<status>                      returned value

  INT

  <status> returns two possible values:

    0 = operation should not be retried.
    1 = operation should be retried.

<filenum>                     input

  INT:value

  is the number of an open file that identifies the file having
  the error.
```

Condition Code Settings

The condition code has no meaning following a call to FILEERROR.

Considerations

- Action of FILEERROR

  The FILEERROR procedure is called after a CCL return from a file system procedure. The FILEERROR procedure determines if an operation should or should not be retried.

  --If the error is caused by one of the following:

  - A normal access request to a terminal currently in BREAK mode
  - BREAK key typed on a terminal where BREAK is enabled
  - Disc pack not up to speed

FILEERROR delays the calling process for one second, then returns a 1, indicating a retry should be performed.

--If the error is an ownership error (error 200) or a path down error (error 201) and the alternate path is operable, FILEERROR returns a 1, indicating that the operation should be retried. If the alternate path is inoperable, a 0 is returned.

--If the error is caused by one of the following:

- A device not ready
- No WRITE ring on a tape unit
- Paper out on a line printer

  An appropriate message is printed on the home terminal and is followed by a READ from the terminal. If STOP is entered after the READ (signaling that the condition cannot be corrected), FILEERROR returns a 0 to indicate that the operation should not be retried. If any other data is entered (typically, carriage return), it signals that the condition has been corrected, and FILEERROR returns a 1 to indicate that the operation should be retried.

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

If the file number has bit <0> set, no message will be printed on the home terminal, unless <filenum> = -1.

To prevent a message from being printed on the home terminal for <filenum> = -1, use <filenum> = %137777.

Example

```
IF FILEERROR ( FNUM ) THEN ... ;              ! retry
```

Related Programming Manual

None

FILEINFO PROCEDURE

The FILEINFO procedure is used to obtain error and characteristic information about a file. The file must be open if you refer to it by its file number, but if you refer to it by its file name, it need not be open.

The syntax for FILEINFO is:

```
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
```

<filenum> input

INT:value

is the number of an open file that identifies the file whose characteristics are to be returned. Either <filenum> or <filename> must be specified; if both are passed, <filename> is the file name associated with <filenum>.

→

NOTE

When <filenum> is specified, the information returned is from the access control block. When <filename> is specified, the information returned is from the file label.

<error> output

INT:ref:1

returns the error number associated with the last operation on the file. <filenum> or <filename> can be specified with this parameter. Refer to the GUARDIAN Operating System Programmer's Guide for error recovery information.

<filename> input, output

INT:ref:12

returns the file name of this file. Either <filenum> or <filename> must be passed; if both are passed, <filename> returns with the name of the file associated with <filenum>. Refer to the GUARDIAN Operating System Programmer's Guide for the file name format.

NOTE

The following parameters are returned with information from the volume directory and not from any system control structures. Subsequently, there is no check to see if the file is actually opened when <filename> is specified: <owner>, <security>, <numextents-allocated>, <maxfilesize>, <partitionsize>, <num-partitions>, and <filetype>.

<ldevnum> output

INT:ref:16 or INT:ref:1

is the logical device number of the device where this file resides. Invalid with <filename>; use <filenum>.

INT:ref:16 is used if the file is partitioned.

INT:ref:1 is used if the file is not partitioned.

→

For partitioned files, an array of <ldevnum> is returned, one entry for each of 16 possible partitions:

```
[0]  = <ldevnum> of partition 0
[1]  = <ldevnum> of partition 1
  .
  .
  .
[15] = <ldevnum> of partition 15.
```

If -1 is returned for a partition, the partition is not open.

<devtype> output

INT:ref:1

returns the device type and subtype of the device associated with this primary partition file. See Appendix B for a list of device types and subtypes.

NOTE

If <devtype>.<0> = 1 this device is a demountable disc volume.

<extent-size> output

INT:ref:1

for disc files returns the primary extent size in 2048-byte units. For nondisc devices, it returns the configured physical record length in bytes. For interprocess files, this parameter has no meaning.

The following parameters apply to disc files only:

<eof-pointer> output

INT(32):ref:1

returns the relative byte address (RBA) of the end-of-file location.

→

```
<next-record-pointer>    output

  INT(32):ref:1

  returns the next-record pointer setting:

     relative files         = a record number
     entry-sequenced files  = a record address
     unstructured files     = an RBA
     key-sequenced files    = parameter is ignored (whatever
                              passes returns unchanged).

  This parameter is not valid with <filename>; use <filenum>.


<last-modtime>           output

  INT:ref:3

  returns a three-word timestamp, indicating the last time the
  file is modified.  <last-modtime> is of the same form as
  the <interval-clock> returned by TIMESTAMP and can be
  converted into a date by CONTIME.


<filecode>               output

  INT:ref:1

  returns the application-defined file code that is assigned
  when the file is created.  File codes 100-999 are reserved
  for use by Tandem.  Refer to the GUARDIAN Operating System
  Utilities Reference Manual for a list of file codes.


<secondary-extent-size>        output

  INT:ref:1

  returns the size of the secondary file extents (extents 1-15)
  in 2048-byte units.
```

→

<current-record-pointer> output

INT(32):ref:1

returns the setting of the current-record pointer. This can be an even or odd value. This parameter is invalid with <filename>; use <filenum>.

relative files	= a record number
entry-sequenced files	= a record address
unstructured files	= an RBA
key-sequenced files	= parameter is ignored (whatever is passed is returned unchanged).

The following parameter applies to any file:

<open-flags> output

INT:ref:1

returns the access granted when the file is opened. This parameter is invalid when used with <filename>; use <filenum>. In this parameter:

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

.<2> = 1 means unstructured access, regardless of the actual file structure (see the OPEN procedure).

.<3:5> is the access mode:

0 = READ/WRITE access
1 = READ-only access
2 = WRITE-only access

.<6> = 1 indicates that resident buffers are provided by the application process for calls to file system I/O routines. a 0 is always returned in this bit (see the OPEN procedure).

.<8> = 1 for process files means, the OPEN message is to be sent nowait and must be completed by a call to AWAITIO.

→

.<9:11> is the exclusion mode:

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

.<12:15> is the maximum number of concurrent nowait I/O operations that can be in progress on this file at any given time. <open flags>.<12:15> = 0 implies wait I/O.

<subdevice> output

INT:ref:1

returns the subdevice number associated with this file. For example, type 61 is an X.25 access method communication line, and subdevice numbers in the range 0-62 can be defined for this line. Note that these values are not used by the operating system. This parameter is invalid with <filename>; use <filenum>.

The following parameters are valid only when used with <filename>:

<owner> output

INT:ref:1

returns the identity of the file's owner in the form <group-num><user-num>.

<security> output

STRING:ref:5

returns the security assigned to the file.

<security>[0].<4> = 1 applies to a program file if the file has PROGID authority. When the program file is called, PROGID sets a caller's accessor ID to the owner ID of the called program file.

<security>[0].<5> = 1	applies if the CLEARONPURGE option is on for this file. If on, this option causes all data to be physically deleted from the disc when the file is purged. If this option is not on, the disc space is only logically deallocated when the file is purged, and no data is actually destroyed.
<security>[1]	returns the reading security of the file.
<security>[2]	returns the writing security of the file.
<security>[3]	returns the execution security of the file.
<security>[4]	returns the purging security of the file.

In <security>[1:4], the returned values are:

returned value:	0	1	2	3	4	5	6	7
security level:	A	G	O	n/a	N	C	U	super

<num-extents-allocated> output

INT:ref:1

returns the number of extents that are allocated for the file.

<max-file-size> output

INT(32):ref:1

returns the maximum number of bytes configured for the file. For example:

<extent-size> + 15 * <secondary-extent-size>

For structured files the value is rounded up to block size.

→

<partition-size> output

INT:ref:1

returns the size of the area needed for each file partition. This file partition information is retrieved from the <partition-parameters> array in the FILERECEIVEINFO procedure.

<num-partitions> output

INT:ref:1

returns the number of partitions configured for the file.

<file-type> output

INT:ref:1

returns the file type and other information about the file. All bits are 0, except as described below:

<file-type>.<2> = 1 for systems with the Transaction Monitoring Facility, indicates this file is audited.

.<10> = 1 indicates REFRESH is specified for this file.

.<11> = 1 for key-sequenced files, indicates index compression is specified.

.<12> = 1 for key-sequenced files, indicates data compression is specified.

= 1 for unstructured files, indicates ODDUNSTR is specified.

.<13:15> specifies the file structure:

0 = unstructured
1 = relative
2 = entry-sequenced
3 = key-sequenced

→

<maximum-extents> output

INT:ref:1

returns the maximum number of extents that can be allocated. This parameter meaningful for a DP2 disc file only.

<unstructured-buffer-size> output

INT:ref:1

returns the internal buffer size to be used for an unstructured file. This parameter is meaningful for a DP2 disc file only.

<open-flags2> output

INT:ref:1

returns various file attribute settings. Unless noted otherwise, the following <open-flags2> bits are meaningful for DP2 disc files only and are valid with both the <filename> and <filenum> parameters:

<open-flags2>.<0> = 0 verify WRITEs off
= 1 verify WRITEs on (current file label default)

.<1> = 0 system automatically selects serial or parallel WRITEs
= 1 serial mirror WRITEs only (current file label default)

.<2> = 0 buffered WRITEs enabled
= 1 WRITE-thru (current file label default)

CAUTION

If the BUFFERED option is specified for an nonaudited file, a system failure or disc-process takeover (with <sync-depth> = 0) could cause the loss of buffered updates for the file that an application might not detect or handle properly unless modified.

.<3> = 0 audit-checkpoint compression off
= 1 audit-checkpoint compression on (current file label default)

⟶

```
                    .<4> = 0  crash-open flag off
                         = 1  crash-open flag on (This is meaningful
                              with the <filename> parameter only
                              and is valid for both DP1 and DP2 disc
                              files.)

                     This is valid for DP1 disc files
                    .<5> = 0  rollforward needed flag off
                         = 1  rollforward needed flag on

                    .<6> = 0  broken file flag off
                         = 1  broken file flag on

                    .<7> = 0  file closed
                         = 1  file opened (This is meaningful with
                              the <filename> parameter only and
                              is valid for both DP1 and DP2 disc
                              files.)

                  .<8:15> = unused
```

<sync-depth> output

INT:ref:1

If this parameter is specified, the <filenum> parameter must be specified and must contain the number of an open file. FILEINFO returns the sync depth (or receive depth for $RECEIVE) of the file. A 0 is returned if the file is not a disc file, process, or $RECEIVE.

<next-open-fnum> output

INT:ref:1

If this parameter is specified, the <filenum> parameter must be specified and must contain the number of an open file or -1.

If an open file number is specified in <filenum>, FILEINFO returns the largest number of an open file whose file number is less than the file number specified in <filenum>. If there is no such file, FILEINFO returns -1.

If -1 is specified in <filenum>, FILEINFO returns the file number of the open file with the largest file number, or -1 if no files are currently open.

Table 2-2 indicates which FILEINFO parameters are valid when specifying the <filenum> or <filename> parameter.

Table 2-2. FILEINFO <filenum> and <filename> Parameters

Parameter	File Number	File Name
([⟨filenum⟩]	x	
,[⟨error⟩]	x	x
,[⟨filename⟩]	x	x
,[⟨ldevnum⟩]	x	x
,[⟨devtype⟩]	x	x
,[⟨extent-size⟩]	x	x
,[⟨eof-location⟩]	x	x
,[⟨next-record-pointer⟩]	x	
,[⟨last-modtime⟩]	x	x
,[⟨filecode⟩]	x	x
,[⟨secondary-extent-size⟩]	x	x
,[⟨current-record-pointer⟩]	x	
,[⟨open-flags2⟩]	x	
,[⟨subdev⟩]	x	
,[⟨owner⟩]		x
,[⟨security⟩]		x
,[⟨num-extents-allocated⟩]		x
,[⟨max-file-size⟩]		x
,[⟨partition-size⟩]		x
,[⟨num-partitions⟩]		x
,[⟨file-type⟩])		x
,[⟨maximum-extents⟩]	x	x
,[⟨unstructured-buffer-size⟩]	x	x
,[⟨open-flags2⟩]	x	x
,[⟨sync-depth⟩]	x	
,[⟨next-open-fnum⟩])	x	

Condition Code Settings

< (CCL) indicates that an error occurred; the error number returns in <error>.

= (CCE) indicates that FILEINFO executed successfully.

> (CCG) does not return from FILEINFO.

Considerations

- Waited OPEN That Failed

  The error number of a preceding AWAITIO on any file or waited OPEN that failed can be obtained by passing a -1 in the <filenum> parameter. The error number returns in <error>.

- Calling FILEINFO Before Opening any Files

  File system error 32 returns in <error> (if the <error> parameter is present in the call) if a process has never opened any files, and -1 is specified in the <filenum> parameter.

- Parameter Exceptions to FILEINFO

  All parameters to FILEINFO are optional except for <filenum> and <filename>, one of which must be given. Placeholder commas must be included to indicate missing parameters, unless the parameters are omitted from the end of the list.

```
CALL FILEINFO ( devicenum, error,,, devicetype,, eof );
```

- Disc File Considerations

  --Finding the error of a preceding CREATE or PURGE that failed

  The error number of a preceding CREATE or PURGE that failed can be obtained by passing a -1 in the <filenum> parameter. The error number returns in <error>.

- Calling FILEINFO Subsequent to a CLOSE

  File system error 16 (file not open) returns if FILEINFO is called subsequent to a CLOSE.

Examples

```
CALL FILEINFO ( FILENUM , ERROR );     ! get error of read
                                       ! operation.

CALL FILEINFO ( , , FILE^NAME , , , , , , , , , , , , , OWNER );
```

Related Programming Manuals

For programming information about the FILEINFO file system procedure, refer to the GUARDIAN Operating System Programmer's Guide and the ENSCRIBE Programming Manual.

FILERECINFO PROCEDURE

The FILERECINFO procedure obtains record characteristics of a disc file.

The syntax for FILERECINFO is:

```
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
```

<filenum> input

INT:value

is the number of an open file that identifies the file whose characteristics are to be returned. You must specify either <filenum> or <filename>; specifying both causes a CCL condition code.

<current-keyspecifier> output

INT:ref:1

returns the current key field's key specifier. This is invalid when you specify the <filename> parameter; use <filenum>.

→

<current-keyvalue> output

STRING:ref:*

returns the value of the current key for <current-keylen> bytes. This is invalid when you specify the <filename> parameter; use <filenum>.

<current-keylen> output

INT:ref:1

returns the current key length in bytes. This is invalid when the <filename> parameter is specified; use <filenum>.

<current-primary-keyvalue> output

STRING:ref:*

returns the value of the current primary key for <current-primary-keylen> bytes. This is invalid when you specify the <filename> parameter; use <filenum>.

<current-primary-keylen> output

INT:ref:1

returns the length, in bytes, of the current primary key. This is invalid when you specify the <filename> parameter; use <filenum>.

<partition-in-error> output

INT:ref:1

returns a number from 0 through 15 that indicates the partition in which the latest error occurred for this file. This is invalid when you specify the <filename> parameter; use <filenum>.

→

<specifier-of-key-in-error> output

INT:ref:1

returns the key tag associated with the latest error occurring with this file. This is invalid when you specify the <filename> parameter; use <filenum>.

The following parameters are the only parameters returned when you specify <filename>:

<file-type> output

INT:ref:1

returns a number indicating the type of file being accessed.

<file-type>.<2>	= 1	for systems with the Transaction Monitoring Facility, indicates this file is audited.
.<10>	= 1	means REFRESH is specified for this file.
.<11>	= 1	for key-sequenced files, means index compression is specified.
.<12>	= 1	for key-sequenced files, means data compression is specified.
	= 1	for unstructured files, means ODDUNSTR is specified.
.<13:15>		specifies the file structure:

0 = unstructured
1 = relative
2 = entry-sequenced
3 = key-sequenced

<logical-recordlen> output

INT:ref:1

returns the maximum size of the logical record in bytes.

→

<blocklen> output

INT:ref:1

returns the length, in bytes, of a block of records for the file.

<key-sequenced-parameters> output

INT:ref:*

is an array to which the parameters unique to a key-sequenced file are returned. (Refer to the CREATE procedure.)

<alternate-key-parameters> output

INT:ref:*

is an array where the parameters describing the file's alternate keys are returned. (Refer to the CREATE procedure.)

<partition-parameters> output

INT:ref:*

is an array where the parameters describing a multivolume file are returned. (Refer to the CREATE procedure.)

<filename> output

INT:ref:12

identifies the file whose characteristics are returned. You must specify either <filenum> or <filename>; specifying both causes a CCL condition code.

When you specify <filename>, the only parameters returned are <filetype>, <logical-recordlen>, <blocklen>, <key-sequenced-parameters>, <alternate-key-parameters>, and <partition-parameters>.

This information is acquired from the volume directory and not from any system control structures, so there is no check to see if the file is actually opened by this or any other process.

Condition Codes

< (CCL) indicates that an error occurred. This can indicate that the specified file was not found or that both <filenum> and <filename> were specified in the same FILERECINFO call.

= (CCE) indicates that FILERECINFO executed successfully.

> (CCG) indicates that the file is not a GUARDIAN disc file.

Example

```
CALL FILERECINFO ( FILE^NUMBER
                 ,                    ! current key specifier.
                 ,                    ! current key value.
                 ,                    ! current key length.
                 ,                    ! current primary key value.
                 ,                    ! current primary key length.
                 ,                    ! partition in error.
                 ,                    ! key in error.
                 ,FILE^TYPE );
```

Related Programming Manual

For programming information about the FILERECINFO file system procedure, refer to the ENSCRIBE Programming Manual.

FIXSTRING PROCEDURE

The FIXSTRING procedure is used to edit a string based on subcommands provided in a template.

The syntax for FIXSTRING is:

```
CALL FIXSTRING ( <template>                          ! i
               ,<template-len>                       ! i
               ,<data>                               ! i, o
               ,<data-len>                           ! i, o
               ,[ <maximum-data-len> ]               ! i
               ,[ <modification-status> ] );         ! o
```

<template> input

STRING:ref:*

is the character string to be used as a modification template.

There are three basic subcommands that you can use in <template>: replacement, insertion, and deletion.

In addition, replacement can be either explicit (a subcommand beginning with "R") or implicit (a subcommand beginning with any nonblank character other than "R," "I," or "D"). The form of <template> is:

<template> = { <subcommand> // ... }

<subcommand> =

```
{ R<replacement string> }     ! replace subcommand
{ I<insertion string>   }     ! insert subcommand
{ D                     }     ! delete subcommand
{ <replacement string>  }     ! implicit replacement
```

<template-len> input

INT:value

is the length, in bytes, of the template string.

→

<data> input, output

STRING:ref:*

on input, is a string to be modified. The resulting string returns in this parameter.

<data-len> input, output

INT:ref:1

on input, contains the length, in bytes, of the string input in <data>. On return, it contains the length, in bytes, of the modified data string in <data>.

<maximum-data-len> input

INT:value

contains the maximum length, in bytes, to which <data> can expand during the call to FIXSTRING. If omitted, 132 is used for this value.

<modification-status> output

INT:ref:1

returns an integer value as follows:

0 = no change was made to <data>.
1 = a replacement, insertion, or deletion was performed on <data> (see "Considerations").

Condition Code Settings

< (CCL) indicates that one or more of the required parameters is missing.

= (CCE) indicates that the operation completed successfully.

> (CCG) indicates that an insert or replace would have caused the <data> string to exceed the <maximum-data-len>.

Considerations

- <template> Considerations

  A character in <template> is recognized as the beginning of a subcommand if it is the first nonblank character in <template>, the first nonblank character following "//," or the first nonblank character following a "D" subcommand. Otherwise, it is considered part of a previous subcommand.

  Note that a subcommand may immediately follow "D" without being preceded by "//."

  If a subcommand begins with "R," "I," or "D," it is recognized as an explicit command. Otherwise, it is recognized as an implied replacement.

  The action of the subcommands is as follows:

  R (or r) for "Replace"

  This subcommand replaces characters in <data> with <replacement-string> on a one-for-one basis. Replacement begins with the character corresponding to R. The <replacement-string> is terminated by the end of <template> or by a "//" sequence in <template>. Trailing blanks are considered part of the replacement string (that is, blanks are not ignored).

  Implied Replacement

  A subcommand that does not begin with "R," "I," or "D" is recognized as a <replacement-string>. Characters in <replacement-string> replace the corresponding characters in <data> on a one-for-one basis.

  D (or d) for "Delete"

  This subcommand deletes the corresponding character in <data>.

  I (or i) for "Insert"

  This subcommand inserts a string from <template> into <data> preceding the character corresponding to the "I". The <insertion-string> is terminated by the end of <template> or by a "//" sequence in <template>. Trailing blanks are considered part of the insertion string (that is, they are not ignored).

- When <data> Is Truncated

  The <maximum-data-len> serves to protect data residing past the end of the <data> string. Therefore, <data> is truncated whenever <data-len> exceeds <maximum-data-len> during processing by FIXSTRING.

In particular, FIXSTRING truncates <data> if <data-len> temporarily exceeds <maximum-data-len>, even if <template> contains delete subcommands that result in a <data> string of the correct length.

- When Insertion String Is Truncated

  If an insertion causes the length of <data> to exceed <maximum-data-len>, the FIXSTRING truncates <insertion-string>.

- <modification-status> is set to 1 if a replacement is performed that leaves <data> unchanged.

Example

```
CALL FIXSTRING ( S^TEMP^ARRAY , TEMP^LEN , SCOMMAND , NUM );
```

Related Programming Manual

For programming information about the FIXSTRING utility procedure, refer to the GUARDIAN Operating System Programmer's Guide.

FL^SCREEN PROCEDURE

For users of the ENTRY or ENTRY520 screen formatter, the FL^SCREEN procedure is called to find the actual length of the data entered.

The syntax for FL^SCREEN is:

```
<length> := FL^SCREEN ( <field-name> );                    ! i

<length>                    returned value

  INT

  is the length of the input data actually typed into the entry
  field.

<field-name>

  STRING:ref:*

  is the name of an entry field.
```

Condition Code Settings

The condition code has no meaning following a call to FL^SCREEN.

Example

```
LENGTH := FL^SCREEN ( X^NAME );
```

Be aware of the difference between the length of the entry field, LENGTH, and the length of the data actually entered, FL^SCREEN (X^NAME).

Related Programming Manuals

For programming information about the FL^SCREEN entry procedure, refer to the ENTRY Screen Formatter Operating and Programming Manual or the ENTRY520 Screen Formatter Operating and Programming Manual.

FNAMECOLLAPSE PROCEDURE

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

The syntax for FNAMECOLLAPSE is:

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


<length>                      returned value

  INT

  returns the number of bytes in <external-name>.


<internal-name>               input

  INT:ref:12

  is the name to be converted.  <internal-name> is an array of
  12 words.  <internal-filename> cannot be the same array as
  <external-filename>.  The file name is one of the following:

    Permanent Disc Files

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

    Temporary Disc Files

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

    Nondisc Devices

       Word [0:11] = $<devname><blank-fill>
                     $<ldevnum><blank-fill>
```

→

```
    Network File Name

       Word [0].<0:7>  = \ (ASCII backslash)
               .<8:15> = <sysnum>, in octal
            [1:3]      = <volname>, <devname>, or <process-id>
            [4:11]     = $<volname><blank-fill>
                                 or
                         $<devname><blank-fill>

  $<volname> in words four through eleven cannot be any longer
  than six characters.


<external-name>                output

  STRING:ref:26 or STRING:ref:34

  returns the external form of <internal-name>.  If
  <internal-name> is a local file name, <external-name> contains
  a maximum of 26 bytes; if a network name is converted,
  <external-name> contains a maximum of 34 bytes. (See the
  FNAMEEXPAND procedure.)
```

Condition Code Settings

The condition code has no meaning following a call to FNAMECOLLAPSE.

Considerations

- Invalid File Names

  It is the responsibility of the program calling FNAMECOLLAPSE to pass a valid file name in <internal-name>. Invalid file names cause unpredictable results such as retrieving information from the wrong file.

- Passing a Bad <sysnum> Value

  If <internal-name> is in network form, and the system number in the second byte does not correspond to any system in the network, FNAMECOLLAPSE supplies "??" as the system name.

Example

```
LENGTH := FNAMECOLLAPSE ( INTNAME , EXTNAME );
```

NOTE

If INTNAME is passed in local internal form for example, "$SYSTEM SUBVOL MYFILE" it converts to external local form, "$SYSTEM.SALES.MYFILE."

If INTNAME is passed in network form for example, "\<sysnum>SYSTEMSUBVOL MYFILE," it converts to external network form, "\<system-name>.$SYSTEM.SUBVOL.MYFILE."

Related Programming Manual

For programming information about the FNAMECOLLAPSE file system procedure, refer to the GUARDIAN Operating System Programmer's Guide.

FNAMECOMPARE PROCEDURE

The FNAMECOMPARE procedure compares two file names within a local or network environment to determine whether these file names refer to the same file or device. For example, one name might be a logical system name or a device number, while the other reference might be a symbolic name. The file names compared must be in the standard 12-word internal format that FNAMEEXPAND returns.

The syntax for FNAMECOMPARE is:

```
{ <status> := } FNAMECOMPARE ( <filename1>          ! i
{ CALL        }                ,<filename2> );      ! i
```

<status> returned value

INT

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

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

```
<filename1>[4] <> <filename2>[4] FOR 8
```

A value less than -1 is the negative of a file system error code. With this error condition, the comparison is not attempted.

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

<filename1> input

INT:ref:12

is the first file name that is compared. Each <filename> array can contain either a local file name or a network file name (see the FNAMECOLLAPSE procedure). Refer to the GUARDIAN Operating System Programmer's Guide for the definitions of file names.

→

<filename2>	input

INT:ref:12

is the second file name that is compared. (See the FNAMECOLLAPSE procedure.)

Condition Code Settings

The value returned from the program function determines the condition code setting. (See the definition of the <status> parameter.)

Considerations

- The arrays containing the file names for comparison are not modified.

- Alphabetic Characters not Upshifted

  Alphabetic characters within qualified process names are not upshifted before comparison.

- Passing Logical Device Numbers for File Names

  If a logical device number format (such as $0076) is used for one file name but not for the second file name, the device table of the referenced system is consulted to determine whether the names are equivalent. This is the only case where the device table is used. All other comparisons involve only the examination of the two file names supplied.

- FNAMECOMPARE and Negative File Errors

  Negative file system error codes indicate that a logical device number format is passed for one file name and not for the second and that the device is connected to a remote network node. Some of the most common negative file system error codes returned are:

  -13 An illegal file name specification for either file name is made.

  -14 The device does not exist. Only one of the file names is passed in logical device number format (requiring a check of the device table), and the file name represents a device connected to a remote node.

-18 No such system is defined in this network. Only one of the file names is passed in logical device number format (requiring a check of the device table), and the file name represents a device connected to a remote node.

-22 A parameter or buffer is out of bounds.

-250 All paths to the system are down. Only one of the file names is passed in logical device number format (requiring a check of the device table), and the file name represents a device connected to a remote node.

Examples

```
FNAME1 ':=' [ "$TERM1" , 9 * [ "  "] ];
FNAME2 ':=' [ %56006 , "TERM1 " , 8 * [ "  "] ];  ! "\" , 6
            , "TERM1";
STATUS := FNAMECOMPARE ( FNAME1 , FNAME2 );
```

Execution of this example returns a 0 in status and the condition code CCE.

In a nonnetwork system, execution of the example returns a status of -1 and the condition code CCL.

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

```
FNAME1 ':=' [ "$SERVR  #START  UPDATING" ];
FNAME2 ':=' [ "$SERVR  #FINISH UPDATING" ];
STATUS := FNAMECOMPARE ( FNAME1 , FNAME2 );
```

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

In any system, execution of

```
FNAME1 ':=' [ "$0013 ", 9 * [ "  "] ];
FNAME2 ':=' [ "$DATAX", 9 * [ "  "] ];
STATUS := FNAMECOMPARE ( FNAME1 , FNAME2 );
```

returns a status of 0 and a condition code CCE if the device name $DATAX is defined as logical device number 13 at SYSGEN time; otherwise, it returns a status of -1 and the condition code CCL.

Related Programming Manual

None

FNAMEEXPAND PROCEDURE

The FNAMEEXPAND procedure is used to expand a partial file name from the compacted external form to the standard 12-word internal form usable by other file system procedures.

The syntax for FNAMEEXPAND is:

```
{ <length> := } FNAMEEXPAND ( <external-filename>          ! i
{ CALL        }               ,<internal-filename>          ! o
                              ,<default-names> );           ! i
```

<length> returned value

INT

returns the length, in bytes, of the file name in <external-filename>. If an invalid file name is specified, 0 is returned.

<external-filename> input

STRING:ref:24 or STRING:ref:34

is the file name to be expanded. The file name must be in the form:

[\<sysname>.]<filename>

where <filename> is in one of these forms:

[$<volname>.][<subvol-name>.]<disc-filename><delim>

$<processname>[.#<1st-qualif-name>[.<2nd-qualif-name>]]<delim>

$<devname><delim>

$<ldevnum><delim>

<delim>

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

⟶

<internal-filename> output

INT:ref:12

is an array of 12 words where FNAMEEXPAND returns the expanded file name. FNAMEEXPAND (unlike FNAMECOLLAPSE) can have the same source and destination buffers (file names) since it uses a temporary intermediate storage area for the conversion. (See "Considerations" for the form of the returned <internal-filename>.)

<default-names> input

INT:ref:8

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

```
<default-names>[0:3] = default <volname> (blank-filled
                       on right)
                 [4:7] = default <subvolname> (blank-filled
                       on right)
                 [0:7] = corresponds directly to <word>[1:8]
                       of the command interpreter startup
                       message.  Refer to the GUARDIAN
                       Operating System Programmer's Guide
                       for the startup message format.
```

Condition Code Settings

The condition code has no meaning following a call to FNAMECOMPARE.

Considerations

- Expanding Network File Names

  --FNAMEEXPAND converts local file names to local names and network file names to network names.

  --When network file names are involved, FNAMEEXPAND converts the system name to the appropriate system number (see "Examples"). (If the system name is unknown, FNAMEEXPAND supplies 255 for the system number.)

- Results of File Name Expansion by FNAMEEXPAND

  <disc-filename> returns as:

```
  <filename>[0:3]  = $<default-volname><blank-fill>
            [4:7]  = <default-subvolname><blank-fill>
            [8:11] = <disc-filename><blank-fill>
```

  <subvolname>.<disc-filename> returns as:

```
  <filename>[0:3]  = $<default-volname><blank-fill>
            [4:7]  = <subvolname><blank-fill>
            [8:11] = <disc-filename><blank-fill>
```

  $<volname>.<disc-filename> returns as:

```
  <filename>[0:3]  = $<volname><blank-fill>
            [4:7]  = <default-subvolname><blank-fill>
            [8:11] = <disc-filename><blank-fill>
```

  $<volname>.<subvolname>.<disc-filename> returns as:

```
  <filename>[0:3]  = $<volname><blank-fill>
            [4:7]  = <subvolname><blank-fill>
            [8:11] = <disc-filename><blank-fill>
```

  $<processname>.#<1st-qualif-name> returns as:

```
  <filename>[0:3]  = $<processname><blank-fill>
            [4:7]  = #<1st-qualif-name><blank-fill>
            [8:11] = <blank-fill>
```

  $<processname>.#<1st-qualif-name>.<2nd-qualif-name> returns as:

```
  <filename>[0:3]  = $<processname><blank-fill>
            [4:7]  = #<1st-qualif-name><blank-fill>
            [8:11] = <2nd-qualif-name><blank-fill>
```

  $<devname> returns as:

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

  $<ldevnum> returns as:

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

  If any of the forms described above are preceded by "\<sysname>," the result is as given above, except that "\<sysnum>" replaces "$" in the result.

  Any other file name is invalid.

Example

```
LENGTH := FNAMEEXPAND ( INNAME , OUTNAME , PSMG[1] );
```

Related Programming Manuals

For programming information about the FNAMEEXPAND file system procedure, refer to the GUARDIAN Operating System Programmer's Guide. For network programming applications, refer to the EXPAND Reference Manual.

FORMATCONVERT PROCEDURE

The FORMATCONVERT procedure converts an external format to internal form for presentation to the FORMATDATA procedure.

The syntax for FORMATCONVERT is:

```
{ <status> := } FORMATCONVERT ( <iformat>              ! i
{ CALL       }                 ,<iformatlen>           ! i
                               ,<eformat>              ! o
                               ,<eformatlen>           ! o
                               ,<scales>               ! o
                               ,<scale-count>          ! o, i
                               ,<conversion> );        ! i
```

<status> returned value

INT

is a value indicating the outcome of FORMATCONVERT:

> 0 indicates successful conversion. The value is the number of bytes in the converted format (<iformat>).

= 0 indicates <iformatlen> was insufficient to hold the entire converted format.

< 0 indicates an error in the format. The value is the negated byte location in the input string at which the error was detected. The first byte of <eformat> is numbered 1.

<iformat> input

STRING:ref:*

is an array in which the converted format is stored. The contents of this array must be passed to the FORMATDATA procedure as an integer parameter, but FORMATCONVERT requires it to be in byte-addressable G-relative storage. Thus <iformat> must be aligned on a word boundary, or the contents of <iformat> must be moved to a word-aligned area when it is passed to FORMATDATA. (The area passed to FORMATDATA need not be in byte-addressable storage.)

→

<iformatlen> input

INT:value

is the length, in bytes, of the <iformat> array. If the converted format is longer than <iformatlen>, the conversion terminates and a <status> value <= 0 returns.

<eformat> output

STRING:ref:*

is the format string in external (ASCII) form.

<eformatlen> output

INT:value

is the length, in bytes, of the <eformat> string.

<scales> output

INT:ref:*

is an integer array. FORMATCONVERT processes the format from left to right, placing the scale factor (the number of digits that appear to the right of the decimal point) specified or implied by each repeatable EDIT descriptor into the next available element of <scales>. This is done until the last repeatable edit descriptor is converted or the maximum specified by <scale-count> is reached, whichever occurs first.

<scale-count> input, output

INT:ref:*

on call, is the number of occurrences of the <scales> array.

On return, <scale-count> contains the actual number of repeatable EDIT descriptors converted.

If the number of repeatable EDIT descriptors present is greater than the number entered here, FORMATCONVERT stops storing scale factors when the <scale-count> maximum is reached, but it continues to process the remaining EDIT descriptors and it continues incrementing <scale-count>.

<conversion> input

INT:value

Specifies the type of conversion to be done:

0 = Check validity of format only. No data is stored into <iformat>. The scale information is stored in the <scale> array.

1 = Produce expanded form with modifiers and decorations. This requires additional storage space, but the execution time is half that of version 2 (below). The size required is approximately 10 times <eformatlen>.

2 = Produce compact conversion, ignoring modifiers and decorations. The resulting format requires little storage space, but the execution time is twice as long as version 1 (above).

NOTE

The <scales> parameter information is included to provide information needed by the ENFORM product. It might not interest most users of FORMATCONVERT. If so, supply a variable initialized to 0 for <scales> and <scale-count>.

Condition Code Settings

The condition code has no meaning following a call to FORMATCONVERT (see the <status> parameter).

Example

```
CALL FORMATCONVERT ( IN^FORMAT , IN^LEN , EXT^FORMAT , EXT^LEN
                   , SCALE , SCALE^CNT , CONVERSION );
```

Related Programming Manuals

For programming information about the FORMATCONVERT procedure, refer to the GUARDIAN Operating System Programmer's Guide.

FORMATDATA PROCEDURE

The FORMATDATA procedure performs conversion between internal and external representations of data, as specified by a format or the list-directed conversion rules.

The syntax for FORMATDATA is:

```
{ <error> := } FORMATDATA ( <buffer>                    ! i, o
{ CALL       }              ,<bufferlen>                ! i
                            ,<buffer-occurs>            ! o
                            ,<length>                   ! o
                            ,<iformat>                  ! o
                            ,<variable-list>            ! o
                            ,<variable-list-len>        ! o
                            ,<flags> );                 ! i
```

<error> returned value

INT:value

indicates the outcome of the call.

0 = successful operation

Formatter errors:

267 = buffer overflow
268 = no buffer
270 = format loopback
271 = EDIT item mismatch
272 = illegal input character
273 = bad format
274 = numeric overflow

<buffer> input, output

STRING:ref:*

is a buffer or a series of contiguous buffers where the formatted output data is placed or where the input data is found. The length, in bytes, of <buffer> must be at least <bufferlen> * <buffer-occurs>.

→

<bufferlen> input

INT:value

is the length, in bytes, of each buffer in the <buffer> array.

<buffer-occurs> input

INT:value

is the number of buffers in <buffer>.

<length> output

INT:ref:*

is an array that must have at least as many elements as there are buffers in the <buffer> array on output. FORMATDATA stores the highest referenced character position in each buffer in the corresponding <length> element. If a buffer is not accessed, -1 is stored for that buffer and for all succeeding ones. If a buffer is skipped (for example, due to consecutive buffer advance descriptors in the format), 0 is stored.

There are no values stored in the <length> parameter during input operation.

<iformat> output

INT:ref:*

is an integer array containing the internal format (as constructed by FORMATCONVERT).

<variable-list> output

INT:ref:*

is a 4- or 5-word entry for each array or variable. Refer to "Considerations" for the contents and form of this array.

→

```
<variable-list-len>              output

  INT:value

  is the number of <variable-list> entries passed in this call.


<flags>                          input

  INT:value

  Bit .<15> = Input

              0 = FORMATDATA performs output operations.
              1 = FORMATDATA performs input operations.

      .<4> = Null value passed

              0 = each <variablelist> item is a 4-word group.
              1 = each <variablelist> item is a 5-word group.

      .<3> = P-Relative (<iformat> array)

              0 = the <iformat> array is G-relative.
              1 = the <iformat> array is P-relative.

      .<2> = List-directed (refer to the GUARDIAN Operating
             System Programmer's Guide for information about
             list-directed operations)

              0 = apply the format-directed operation.
              1 = apply the list-directed operation.
```

Condition Code Settings

The condition code has no meaning following a call to FORMATDATA (see the <error> parameter).

Considerations

- A passed P-relative array must be in the same space as the call.
- <variable-list> Array Form

  The 4- or 5-word entry for each array or variable consists of the following items:

Word	Contents
[0]	dataptr
[1]	datatype
[2]	databytes
[3]	dataoccurs
[4]	nullptr (optional)

<dataptr>

is the address of the array or variable (byte address for types 0, 1, 12-15, and 17 and word address for other types).

<datatype>

is the type and scale factor of the element:

bits <8:15>

```
 0 = string
 1 = numeric string unsigned
 2 = integer(16) signed
 3 = integer(16) unsigned
 4 = integer(32) signed
 5 = integer(32) unsigned
 6 = integer(64) signed

 7 = not used
 8 = real(32)
 9 = complex(32*2)
10 = real(64)
11 = complex(64*2)
12 = numeric string, sign trailing, embedded
13 = numeric string, sign trailing, separate
14 = numeric string, sign leading, embedded
15 = numeric string, sign leading, separate

16 = not used
17 = logical * 1 (1 byte)
```

18 = not used
19 = logical * 2 (INT(16))

20 = not used
21 = logical * 4 (INT(32))

NOTE

Data types 7 through 11 require floating-point firmware.

bits <0:7> Scale factor moves the position of the implied decimal point by adjusting the internal representation of the expression. Scale factor is the number of positions that the implied decimal point is moved to the left (factor > 0) or to the right (factor <= 0) of the least significant digit. This value must be 0 for data types 0, 17, 19, and 21.

<databytes>

is the size, in bytes, of the variable or array element used to determine the size of strings and address spacing.

<dataoccurs>

is the number of elements in the array (supply 1 for undimensioned variables).

<nullptr>

If <> 0, it is the byte address of the null value.
If = 0, there is no null value for this variable.

Example

```
ERROR := FORMATDATA ( BUFFERS , BUF^LEN , NUM^BUFS , BUF^LENS
                    , WFORMAT , VLIST , 4 , 0 );
```

Related Programming Manual

For programming information about the FORMATDATA procedure, refer to the GUARDIAN Operating System Programmer's Guide.

GETCPCBINFO PROCEDURE

The GETCPCBINFO provides a process with information from its own (the current) process control block (PCB).

The syntax for GETCPCBINFO is:

```
CALL GETCPCBINFO ( <request-id>                    ! i
                  ,<cpcb-info>                     ! o
                  ,<in-length>                     ! i
                  ,<out-length>                    ! o
                  ,<error>  );                     ! o
```

<request-id> input

INT:value

specifies the information to be returned. The list of valid request IDs are:

0 = remote creator flag; returns 1 in <cpcb-info> if creator was remote.
1 = logged-on process state; returns 1 in <cpcb-info> if the process has logged on.

<cpcb-info> output

INT:ref:*

is an array that returns with the information requested from the PCB.

<in-length> input

INT:value

specifies the length, in bytes, of the <cpcb-info> array. (This is used to prevent possible data overrun.)

<out-length> output

INT:ref:1

specifies the number of bytes of information returned in <cpcb-info>.

<error> output

INT:ref:1

returns a file system error number indicating the outcome of the PCB information request.

Condition Code Settings

The condition code has no meaning following a call to GETCPCBINFO.

Example

```
CALL GETCPCBINFO ( REQUEST^ID
                 , PCB^INFO
                 , IN^LENGTH
                 , OUT^LENGTH
                 , ERROR^REQUEST );
```

Related Programming Manual

None

GETCRTPID PROCEDURE

The GETCRTPID procedure is used to obtain the CRTPID (which is the process name in words[0:2] or creation timestamp and two blanks or CPU and pin in word[3]) associated with a process. (Refer to the GUARDIAN Operating System Programmer's Guide for more information about process IDs (PIDs) and CRTPIDs.)

The syntax for GETCRTPID is:

```
CALL GETCRTPID ( <cpu,pin>                              ! i
                ,<process-id> );                        ! o
```

<cpu,pin> input

INT:value

is the processor and pin number of the process whose CRTPID is returned.

<process-id> output

INT:ref:4

is an array of four words where GETCRTPID returns the CRTPID of the specified processor and pin number. The PID is in local form. Refer to the GUARDIAN Operating System Programmer's Guide for information about PIDs.

The PID is returned in local form (that is, $<process-name> is in word[0:2], and word[3] contains the <cpu,pin>).

Condition Code Settings

< (CCL) indicates that GETCRTPID failed, or that no <pid> exists.

= (CCE) indicates that GETCRTPID completed successfully.

> (CCG) does not return from GETCRTPID.

Considerations

- Passing the PID to OPEN

  The PID returned from GETCRTPID is suitable for passing directly to the file system OPEN procedure (if blank-filled on the right).

- An Application Acquiring Its Own PID

  An application can acquire its own PID by passing the results of the MYPID procedure to the GETCRTPID procedure:

```
CALL GETCRTPID ( MYPID , MY^PROCESSID );
```

Example

```
CALL GETCRTPID ( PID , PROCESS^ID );
```

Related Programming Manual

None

GETDEVNAME PROCEDURE

The GETDEVNAME procedure is used to obtain the name associated with a logical device number. GETDEVNAME returns the name of a designated logical device, if such a device exists, or the name of the next higher (numerically) logical device if the designated logical device does not exist. A status word is returned from GETDEVNAME that indicates whether or not the designated device exists or if a higher entry exists. By repeatedly calling GETDEVNAME and supplying successively higher logical device numbers, you can obtain the names of all system devices.

The syntax for GETDEVNAME is:

```
<status> := GETDEVNAME ( <ldevnum>              ! i, o
                        ,<devname>              ! o
                        ,[ <sysnum> ]  );       ! i
```

<status> returned value

INT

indicates the outcome of the call. The values of <status> can be:

- 0 = successful; the name of the designated logical device is returned in <devname>.
- 1 = the designated logical device does not exist. The logical device number of the next higher device is returned in <ldevnum>; the name of that device is returned in <devname>.
- 2 = there is no logical device with <ldevnum> equal to or greater than <ldevnum>.
- 4 = the system specified could not be accessed.
- 99 = parameter error.

<ldevnum> input, output

INT:ref:1

is the logical device number of the designated logical device whose name is returned.

→

On return, <ldevnum> receives the number of the logical device, if one exists, or the number of the next higher (numerically) logical device. The <ldevnum> remains unchanged if no higher logical device exists.

<devname> output

INT:ref:4

returns the device name or volume name of the designated device, if it exists, or the next higher (numerically) logical device if the designated device does not exist. The <ldevname> remains unchanged if no higher logical device exists.

<sysnum> input

INT:value

specifies the system (in a network) that is searched for <ldevnum>. If omitted, the local system is assumed.

Condition Code Settings

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

Considerations

- When Device Name Is Returned in Network Form

  If the device specified by <ldevnum> is remote, its device name is returned in network form. If the device specified by <ldevnum> is local, the device name is returned in local form.

- <sysnum> and Device Names Containing Seven Characters

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

- A process name is returned as a device name if you specify a logical device number that corresponds to a destination control table (DCT) entry for a process.

- When Zeros and Blanks Are Returned

  Zeros and blanks are returned if the disc is demounted, or if the controlling CPUs are not up.

Example

```
STATUS := GETDEVNAME ( LDEVNUM , DEVNAME , SYSNUM );
```

Related Programming Manual

None

GETPOOL PROCEDURE

The GETPOOL procedure obtains a block of memory from a buffer pool.

The syntax for GETPOOL is:

```
<address> := GETPOOL ( <pool-head>                ! i, o
                      ,<block-size> );            ! i
```

<address> returned value

INT(32)

returns the extended address of the memory block obtained if the operation is successful or -1D if an error occurred or <block-size> is 0.

WARNING

<address> should be a simple INT(32) variable; otherwise, the assignment can alter the condition code.

<pool-head> input, output

INT .EXT:ref:19

is the pool head previously defined by a call to DEFINEPOOL.

<block-size> input

INT(32):value

is the size, in bytes, of the memory obtained from the pool. This number cannot be greater than %377770D. To check data structures without getting any memory from the pool, set <block-size> to zero.

Condition Code Settings

<	(CCL)	indicates that <block-size> is out of range, or that the data structures are invalid; -1D is returned.

= (CCE) indicates that the operation is successful; extended address of block returns if <block-size> is greater than zero, or -1D returns if <block-size> is equal to 0.

> (CCG) indicates that insufficient memory is available; -1D returns.

Considerations

- A Bounds Violation Trap

  GETPOOL and PUTPOOL do not check pool data structures on each call. A process that destroys data structures can get a bounds violation trap on a call to GETPOOL or PUTPOOL.

Example

```
@PBLOCK := GETPOOL( POOL^HEAD , $UDBL( $LEN( PBLOCK ) );
    ! get pool block size of PBLOCK in bytes.
```

Related Programming Manual

For programming information about the GETPOOL memory management procedure, refer to the GUARDIAN Operating System Programmer's Guide.

GETPPDENTRY PROCEDURE

The GETPPDENTRY procedure is used to obtain a description of a named process pair by its index into the destination control table (DCT). To obtain process pair descriptions by process name, use LOOKUPPROCESSNAME procedure.

The syntax for GETPPDENTRY is:

```
CALL GETPPDENTRY ( <index>                          ! i
                 ,<sysnum>                          ! i
                 ,<ppd>    );                       ! o
```

<index> input

INT:value

specifies which DCT entry returns. The first entry is 0, the second is 1, etc.

<sysnum> input

INT:value

specifies the system where the process pair exists.

<ppd> output

INT:ref:9

is an array where GETPPDENTRY returns the nine-word DCT entry specified by the given <index> and <sysnum>. Its format is:

```
  <ppd>       [0:2]      = process name (in local form)

              [3].<0:7>  = <cpu> of primary process
                 .<8:15> = <pin> of primary process
```

→

```
[4].<0:7>   = <cpu> of backup process if it is a
              process pair.  (This is 0 if
              there is no backup.)
   .<8:15>  = <pin> of backup process, if it is a
              process pair.  (This is 0 if there
              is no backup.)

[5:8]       = <process-id> of ancestor.
```

Condition Code Settings

< (CCL) indicates that the DCT in the given system cannot be accessed.

= (CCE) indicates that the GETPPDENTRY completed successfully.

> (CCG) indicates that the <index> is greater than the last entry in the DCT.

Considerations

- Checking the DCT Entry

  If <index> is not currently being used, GETPPDENTRY returns CCE and sets <ppd> to zeros. To check for all conditions, an application could contain the following code:

```
CALL GETPPDENTRY( INDEX^NUM , SYS^NUM , PROCESS^PAIR^DESCRIPT );
IF < THEN ... ; ! no more entries or system unavailable.
IF = AND PROCESS^PAIR^DESCRIPT THEN ... ! found an entry.
ELSE
  ! unused entry, try the next INDEX^NUM.
```

- Difference Between GETPPDENTRY and LOOKUPPROCESSNAME

  The difference between the GETPPDENTRY procedure and the LOOKUPPROCESSNAME procedure is:

  GETPPDENTRY is primarily used to obtain a local or remote process pair description by its index into a system table.

LOOKUPPROCESSNAME is primarily used to obtain a local or remote process pair description by its name.

Example

See "Considerations."

Related Programming Manual

None

GETREMOTECRTPID PROCEDURE

The GETREMOTECRTPID procedure returns the CRTPID (which is the process name in words[0:2] or creation timestamp and two blanks or CPU and pin in word[3]) of a remote process whose processor, pin, and system number are known.

The syntax for GETREMOTECRTPID is:

```
CALL GETREMOTECRTPID ( <cpu,pin>                    ! i
                      ,<process-id>                 ! o
                      ,<sysnum> );                  ! i
```

<cpu,pin> input

INT:value

is the processor and pin number of the process whose CRTPID is to be returned.

<process-id> output

INT:ref:4

is an array of four words where GETREMOTECRTPID returns the CRTPID of the processor and pin.

If <sysnum> specifies a remote system, the process ID (PID) is in network form; if <sysnum> specifies the local system, the PID is in local form. Refer to the GUARDIAN Operating System Programmer's Guide for information about PID.

<sysnum> input

INT:value

is a value specifying the system for which the CRTPID is found.

Condition Code Settings

< (CCL) indicates the GETREMOTECRTPID failed for one of the following reasons:

- no such process exists
- the remote system could not be accessed
- the process has an inaccessible name, consisting of more than four characters.

= (CCE) indicates that GETREMOTECRTPID was successful.

> (CCG) does not return from GETREMOTECRTPID.

Example

```
CALL GETREMOTECRTPID ( PID , CRT^PID , SYS^NUM );
```

Related Programming Manual

None

GETSYNCINFO PROCEDURE

The GETSYNCINFO procedure is called by the primary process of a primary or backup process pair before starting a series of WRITE operations to a file open with paired access. GETSYNCINFO returns a disc file's synchronization block so that it can be sent to the backup process in a checkpoint message.

NOTE

Typically, GETSYNCINFO is not called directly by application programs. Instead, it is called indirectly by CHECKPOINT.

The syntax for GETSYNCINFO is:

```
CALL GETSYNCINFO ( <filenum>                          ! i
                  ,<sync-block>                       ! o
                  ,[ <sync-block-size> ] );           ! o
```

<filenum> input

INT:value

is the number of an open file that identifies the file whose sync block is obtained.

<sync-block> output

INT:ref:*

returns the synchronization block for this file. The size, in words, of <sync-block> is determined as follows:

- For unstructured disc files, size = 4 words.
- For ENSCRIBE structured files, size in words = 11 + (longest alt key len + pri key len + 1) / 2.
- For the Transaction Monitoring Facility, the transaction pseudofile size = 9 words.
- For processes, size = 2 words.

<sync-block-size> output

INT:ref:1

returns the size, in words, of the sync block data.

Condition Code Settings

< (CCL) indicates that an error occurred (call FILEINFO).

= (CCE) indicates that GETSYNCINFO was successful.

> (CCG) indicates that the file is not a disc file.

Considerations

- File Number Has Not Been Opened

  If the GETSYNCINFO file number does not match the file number of the open file that you are trying to access, then the call to GETSYNCINFO returns with file system error 16.

- Buffer Address Out of Bounds

  If an out-of-bounds application buffer address parameter is specified in the GETSYNCINFO call (that is, a pointer to the buffer has an address that is greater than the MEM associated with the data area of the process) or if the buffer lies within the data area that is used by GETSYNCINFO, then the call returns with file system error 22.

Example

```
CALL GETSYNCINFO ( FILE^NUM , SYNC^ID );
```

Related Programming Manual

For programming information about the GETSYNCINFO checkpointing procedure, refer to the GUARDIAN Operating System Programmer's Guide.

GETSYSTEMNAME PROCEDURE

The GETSYSTEMNAME procedure supplies the system name associated with a system number.

The syntax for GETSYSTEMNAME is:

```
{ <ldev> := } GETSYSTEMNAME ( <sysnum>                  ! i
{ CALL      }                 ,<sysname> );             ! o


<ldev>                        returned value

  INT

  if positive, returns the logical device number of the network
  line handler that controls the current path to the system
  designated by <sysnum>.  Other possible returns are:

     0 =  the <sysnum> is not defined.
    -1 =  all paths to the system are down.
    -2 =  system not on a network or the system is local
          and unnamed (system name is blank).

<sysnum>                      input

  INT:value

  {0:254} is the number of the system whose name returns
  into <sysname>.


<sysname>                     output

  INT:ref:4

  returns the name of the system corresponding to <sysnum>.
```

Condition Code Settings

The condition code has no meaning following a call to GETSYSTEMNAME.

Considerations

- If the local system is not part of a network and is not named, then

```
CALL GETSYSTEMNAME ( MYSYSTEMNUMBER ,NAME );
```

returns all blanks to "NAME."

Example

```
LDEV := GETSYSTEMNAME( SYS^NUM ,SYS^NAME );
```

Related Programming Manual

None

GETTMPNAME PROCEDURE

GETTMPNAME obtains the logical device name of the Transaction Monitoring Process (TMP) that was used during system configuration. Opening the transaction pseudofile (TFILE) requires use of that symbolic name, which normally is $TMP.

The syntax for GETTMPNAME is:

```
<status>  := GETTMPNAME ( <devname> );                    ! o
```

<status> returned value

INT

returns a zero, if the call was successful, or a file system error number. (Refer to the System Messages Manual for a list of all file system errors).

<devname> output

INT:ref:12

is an array of 12 words to which GETTMPNAME returns the dummy device name of the TMP. If the TMP is not configured, the device name is all blanks.

Condition Code Settings

The condition code has no meaning following a call to GETTMPNAME.

Considerations

- A process can open the TFILE only once. Attempting to open the TFILE more than once causes a file system error 12. The OPEN TFILE call must be done before the first call to BEGINTRANSACTION, because BEGINTRANSACTION implicitly opens the TFILE, if the process wishes to have concurrent active transactions.

- Out-of-Bounds Parameter or Buffer Address

  If an out-of-bounds application parameter or buffer address parameter is specified--that is, a pointer to the buffer has an address that is greater than the MEM associated with the data area of the process--then the call is rejected and returns file system error 22.

Example

```
ERR^CODE := GETTMPNAME ( TMP^DUMMY^NAME );
```

Related Programming Manual

For programming information about the GETTMPNAME procedure, refer to the Transaction Monitoring Facility (TMF) Reference Manual.

GETTRANSID PROCEDURE

GETTRANSID returns the current transaction identifier of the calling process.

The syntax for GETTRANSID is:

```
<status> := GETTRANSID ( <transid> );                    ! o
```

<status> returned value

INT

returns a 0, if successful, or a file system error number. (Refer to the System Messages Manual for a list of all file system errors.)

<transid> output

INT:ref:4

is an array of four words to which GETTRANSID returns the current-transaction identifier. Its form is:

transid[0].<0:7>	contains 1 plus the EXPAND system number of the system in which BEGINTRANSACTION is 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 is called.
transid[1-2]	contains a double-word sequence number to make the transaction identification unique.
transid[3]	contains a crash count indicating the number of times the home node has had a total system failure since the last time the TMFCOM command INITIALIZE TMF was issued on the home node.

Condition Code Settings

The condition code has no meaning following a call to GETTRANSID.

Considerations

- Out-of-Bounds Parameter or Buffer Address

  If an out-of-bounds application parameter or buffer address parameter is specified--that is, a pointer to the parameter or buffer has an address that is greater than the MEM associated with the data area of the process--the call is rejected and returns file system error 22.

- Requesting Process and Current-Transaction Identifier

  If the requesting process has no current-transaction identifier the call is rejected and returns file system error 75.

- When GETTRANSID Fails

  GETTRANSID fails if TMF is not running on the local system, or I/O fails if TMF is not running on the remote system. In either case, the call returns file system error 82.

Example

```
STATUS := GETTRANSID ( TRANS^ID );
```

Related Programming Manual

For programming information about the GETTRANSID procedure, refer to the Transaction Monitoring Facility (TMF) Reference Manual.

HALTPOLL PROCEDURE

The HALTPOLL procedure is normally used to stop continuous polling.

The syntax for HALTPOLL is:

```
CALL HALTPOLL ( <filenum> );                    ! i

<filenum>                    input

  INT:value

  is the number of an open file which HALTPOLL stops, that
  caused a particular communication line to open.
```

Condition Code Settings

<	(CCL)	indicates that an error occurred (call FILEINFO).
=	(CCE)	indicates that the HALTPOLL procedure executed successfully.
>	(CCG)	does not return from HALTPOLL.

Example

```
CALL HALTPOLL ( FNUM );
```

FNUM is the integer specified in the OPEN call that opens the particular communication line. HALTPOLL forces the immediate termination of an outstanding nowait READ operation within a point-to-point station, or it stops any polling that is in progress within a multipoint station.

Related Programming Manuals

For programming information about the HALTPOLL system procedure, refer to the data-communication manuals.

HEAPSORT PROCEDURE

The HEAPSORT procedure is used to sort an array of equal-sized elements in place.

The syntax for HEAPSORT is:

```
CALL HEAPSORT ( <array>                                  ! i, o
              ,<num-elements>                            ! i
              ,<size-of-element>                         ! i
              ,<compare-proc> );                         ! i
```

<array> input, output

INT:ref:*

contains equal-sized elements to be sorted.

<num-elements> input

INT:value

is the number of elements in <array>.

<size-of-element> input

INT:value

is the size, in words, of each element in <array>.

<compare-proc> input

INT PROC

is an application-supplied function procedure that HEAPSORT calls to determine the sorted order (ascending or descending) of the elements in <array>. It must be of the form:

```
INT PROC <compare-proc> ( <element-a> , <element-b> )
       INT .<element-a>;
       INT .<element-b>;
```

→

The <compare-proc> must compare <element-a> with <element-b> and return either of the following values:

0 (indicating false) if <element-b> should precede <element-a>

1 (indicating true) if <element-a> should precede <element-b>

<element-a> and <element-b> are INT:ref parameters.

Condition Code Settings

The condition code has no meaning following a call to HEAPSORT.

Example

```
CALL HEAPSORT ( ARRAY , NUM^ELEMENTS , ELEMENT^SIZE , ASCENDING );

!  sorts the elements in "array" in ascending order!
```

Related Programming Manual

For programming information about the HEAPSORT utility procedure, refer to the GUARDIAN Operating System Programmer's Guide.

INITIALIZER PROCEDURE

The INITIALIZER procedure is used to read the startup message and, optionally, to request assign and param messages sent by the GUARDIAN Command Interpreter. The INITIALIZER procedure optionally prepares tables of a predefined structure and properly initializes file control blocks (FCBs) with the information read from the startup and assign messages.

The syntax for INITIALIZER is:

```
{ <status> := } INITIALIZER ( [ <rucb> ]                ! i
{ CALL       }                ,[ <passthru> ]           ! o
                              ,[ <startupproc> ]        ! i
                              ,[ <paramsproc> ]         ! i
                              ,[ <assignproc> ]         ! i
                              ,[ <flags> ]  );          ! i
```

<status> returned value

INT

returns one of the following values:

0 = This is the primary process (of a potential process pair).

-1 = If this is the backup process, and CHECKMONITOR returned (indicating that the primary failed before establishing a takeover point), and bit 12 of <flags> is 1.

<rucb> input

INT:ref:*

is a table containing pointers to the FCBs (see "Considerations").

→

<passthru> output

INT:ref:*

is an array where the <startupproc>, <assignproc>, and <paramsproc> procedures can return information to the caller of the INITIALIZER.

<startupproc>,<paramsproc>,<assignproc> input

are application-supplied message-processing procedures that the INITIALIZER calls when it receives a message of the appropriate type.

These procedures must be of the form:

```
PROC <name> ( [ <rucb> ]
              ,[ <passthru> ]
              ,[ <message> ]
              ,[ <msglen> ]
              ,[ <match> ] )
              VARIABLE
```

<rucb>

INT:ref:*

is the run-unit control block described in the GUARDIAN Operating System Programmer's Guide.

<passthru>

INT:ref:*

is an array where the procedure can save information for the caller of the INITIALIZER.

<message>

INT:ref:*

is the startup, the param, or one of the assign messages received. The maximum length of a message is 1028 bytes (including the trailing null characters).

→

```
      <msglen>

        INT:value

        is the length, in bytes, of the message.

      <match>

        INT:value

        is the match count.  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 files' (or file's)
        FCBs, and the match count is incremented.

        If this is not an assign message or if the <rucb>
        parameter is not passed, the match count is always
        0.

<flags>                          input

  INT:value

  contains several fields that determine actions to be taken
  by the INITIALIZER, as follows:

    <flags>.<0:10> = must be 0
           .<11>   = request assign and param messages?
                     0 = yes, 1 = no
           .<12>   = abnormally end if backup takeover occurs
                     before first primary stack checkpoint?
                     0 = yes, 1 = no
           .<13>   = if 1, CALL MONITORNET (-1).
           .<14>   = if 1, CALL MONITORCPUS (-1).
           .<15>   = if 1, CALL ARMTRAP (-1,-1).
```

Condition Code Settings

The condition code has no meaning following a call to INITIALIZER.

Considerations

- $RECEIVE and the INITIALIZER Procedure

  The INITIALIZER procedure provides a way of receiving startup, assign, and param messages without concern for details of the $RECEIVE protocol. (Refer to the GUARDIAN Operating System Programmer's Guide for information about $RECEIVE.) The INITIALIZER opens and obtains messages from $RECEIVE; calls the user-supplied procedure, passing the messages as a parameter to the procedure; and closes $RECEIVE.

- Sequential I/O Procedures and FCBs

  If the <rucb> parameter is supplied, the INITIALIZER stores FCBs based on the information supplied by the startup and assigns messages. These FCBs are in the form expected by the sequential I/O (SIO) procedures and can be used with the SIO procedures without change. If the application does not use the SIO procedures to access the files, the information recovered from the assign messages can be obtained from the FCBs by using the SET^FILE procedure. Refer to the GUARDIAN Operating System Programmer's Guide for additional information about SIO procedures.

- Assign and Param Messages

  When invoked by the primary of a potential process pair, 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 (or files') FCB(s), 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 procedure is useful for the SIO procedures (see Section 3).

Related Programming Manual

For programming information about the INITIALIZER utility procedure, refer to the GUARDIAN Operating System Programmer's Guide.

INTERPRETJULIANDAYNO PROCEDURE

The INTERPRETJULIANDAYNO procedure converts a Julian day number to the year, month, and day.

The Julian calendar is the integral number of days since January 1, 4713 B.C. The formal definition of the Julian day number states that it starts at noon, Greenwich mean time (GMT). For simplicity, we assume the Julian day number starts at midnight, local or Greenwich time, depending on the base of the timestamp.

The syntax for INTERPRETJULIANDAYNO is:

```
CALL INTERPRETJULIANDAYNO ( <julian-day-num>          ! i
                          ,<year>                     ! o
                          ,<month>                    ! o
                          ,<day>   );                 ! o
```

<julian-day-num> input

INT(32):value

is the Julian day number to be converted. The <julian-day-num> must be greater than or equal to 1,721,426 (January 1, 0001) or an arithmetic-overflow trap occurs.

<year> output

INT:ref:1

returns the Gregorian year (for example, 1984, 1985, and so forth).

<month> output

INT:ref:1

returns the Gregorian month (1-12).

<day> output

INT:ref:1

returns the Gregorian day of the month (1-31).

Condition Code Settings

The condition code has no meaning following a call to INTERPRETJULIANDAYNO.

Example

```
CALL INTERPRETJULIANDAYNO ( JULIANDAYNO , YR , MN , DAY );
```

Related Programming Manual

For programming information about the INTERPRETJULIANDAYNO procedure, refer to the GUARDIAN Operating System Programmer's Guide.

INTERPRETTIMESTAMP PROCEDURE

The INTERPRETTIMESTAMP converts a 64-bit Julian timestamp into a Gregorian (the common civil calendar) date and time of day.

The syntax of INTERPRETTIMESTAMP is:

```
<ret-date-time> := INTERPRETTIMESTAMP ( <julian-timestamp>    ! i
                                      , <date-n-time>  );    ! o
```

<ret-date-time> returned value

INT(32)

returns the 32-bit Julian day number.

<julian-timestamp> input

FIXED:value

is a Julian four-word timestamp to be converted.

<date-and-time> output

INT:ref:8

returns an array containing a date and the time of day. This array has the following form:

```
<date-n-time>[0] = the Gregorian year        (1984, 1985, ... )
             [1] = the Gregorian month                 (1-12)
             [2] = the Gregorian day of month          (1-31)
             [3] = the hour of the day                 (0-23)
             [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)
```

Condition Code Settings

The condition code has no meaning following a call to INTERPRETTIMESTAMP.

Considerations

- No checking is performed for the range of the Julian timestamp. The caller must check that the Julian timestamp corresponds to a time in the range of 1 January 0001 00:00 to 31 December 4000 23:59:59.999999.

Example

```
RETURN^DATE^TIME := INTERPRETTIMESTAMP (JULIAN^TIME , DATE^TIME );
```

Related Programming Manual

For programming information about the INTERPRETTIMESTAMP procedure, refer to the GUARDIAN Operating System Programmer's Guide.

JULIANTIMESTAMP PROCEDURE

The JULIANTIMESTAMP procedure returns a four-word, microsecond resolution, JULIAN-date-based timestamp.

The Julian calendar is the integral number of days since January 1, 4713 B.C. The formal definition of the Julian day number states that it starts at noon, Greenwich mean time (GMT). For simplicity, we assume the Julian day number starts at midnight.

The caller can select Greenwich time.

The syntax for JULIANTIMESTAMP is:

```
<retval> := JULIANTIMESTAMP ( [ <type> ]                    ! i
                             ,[ <tuid> ]  );                ! o
```

<retval> returned value

FIXED

is a value representing the number in microseconds since B.C. 4713 January 1, 12:00 (JULIAN proleptic calender). To convert the <retval> to a more usuable form, use the INTERPRETTIMESTAMP procedure.

<type> input

INT:value

is one of the following values specifying the type of time requested:

- 0 = current GMT
- 1 = system-load GMT
- 2 = SYSGEN GMT

If <type> is out of range (that is, not 0 or 1) then 0 is used. <type> 2 is not available for the B00 version of the operating system.

→

<tuid>	output
INT:ref:1 is a time-update ID. This is used when calling the SETSYSTEMCLOCK procedure with relative GMT (see the SETSYSTEMCLOCK procedure).	

Condition Code Settings

The condition code has no meaning following a call to JULIANTIMESTAMP.

Considerations

- System message -10 allows processes to determine the magnitude of and the reason for a time change. Refer to Appendix F for descriptions of interprocess system messages sent to processes.

Example

```
CALL JULIANTIMESTAMP ( TIME^STAMP );
```

Related Programming Manual

For programming information about the JULIANTIMESTAMP procedure, refer to the GUARDIAN Operating System Programmer's Guide.

KEYPOSITION PROCEDURE

The KEYPOSITION procedure is used to position by primary key within key-sequenced files and to position by alternate key within key-sequenced, relative, and entry-sequenced files.

KEYPOSITION sets the current position, access path, and positioning mode for the specified file. The current position, access path, and positioning mode define a subset of the file for subsequent access.

The syntax for KEYPOSITION is:

```
CALL KEYPOSITION ( <filenum>                          ! i
                  ,<key-value>                        ! i
                  ,[ <key-specifier> ]                ! i
                  ,[ <length-word> ]                  ! i
                  ,[ <positioning-mode> ] );          ! i
```

<filenum> input

INT:value

is the number of an open file where the positioning is to take place.

<key-value> input

STRING:ref:*

is the value that defines the current position in the file. The current position is found by a search of the access path specified by <key-specifier>. The first record having an access path key-field value that matches <key-value>, as defined by <positioning-mode> and <compare-length>, becomes the current position.

→

<key-specifier> input

INT:value

designates the key field to be used as the access path for the file:

<key-specifier> = 0 or if omitted, means use the file's primary key as the access path.

= predefined key specifier for an alternate-key field means use that field as the access path.

<length-word> input

INT:value

contains two values:

<length-word>.<0:7> = <compare-length> (left byte)
.<8:15> = <key-length> (right byte)

.<0:7> (<compare-length>) specifies, in bytes, the difference between the length of <key-value> and the specified key field in the file.

.<8:15> (<key-length> specifies how many bytes of the <key-value> are to be searched for in the file to find the initial position.

- If <length-word> is omitted, <compare-length> and <key-length> are defined to be the length of the key (<key-specifier>) defined when the file was created. That is, if <key-specifier> is omitted or 0, <compare-length> and <key-length> are the length of the primary key. If <key-specifier> is the key specifier for an alternate key, the length of the alternate-key field is used.

- If <length-word> is 0, <compare-length> and key-length> are also 0. This results in positioning to the beginning of the file. (Although <key-value> is still a required parameter, its value and that of <positioning-mode> are ignored when <length-word> = 0.)

- If <key-length> = 0 and <compare-length> <> 0, file system error 21 is returned from KEYPOSITION.

→

- If <key-length> <> 0 and <compare-length> = 0, <compare-length> is defined to be the minimum of <key-length> or the key length defined when the file was created.

- If <key-length> <> 0 and <compare-length> <> 0, 0 is used for both values.

See "KEYPOSITION and File System Error 21" under "Considerations".

<positioning-mode> input

INT:value

<positioning-mode>.<0> if 1, and a record with exactly the key specified is found, the record is skipped.

.<14:15> indicates the type of key search to perform and the subset of records obtained.

0 = approximate

Positioning occurs to the first record whose key field, as designated by the <key-specifier>, contains a value equal to or greater than <key-value> for <compare-length> bytes.

1 = generic

Positioning starts at the first record whose key field, as designated by the <key-specifier>, contains a value equal to or greater than <key-value> for <key-length> bytes. Records will be accessed according to whose key field contains a value equal to <key-value> for <compare-length> bytes.

→

2 = exact

Positioning occurs to the first record whose key field, as designated by the <key-specifier>, contains a value of exactly <compare-length> bytes and is equal to <key-value>.

If <positioning-mode> is omitted, the approximate mode is used.

Condition Code Settings

< (CCL) indicates that an error occurred (call FILEINFO).

= (CCE) indicates that the KEYPOSITION was successful.

> (CCG) indicates there is no operation: this is not a structured disc file.

Considerations

- The calling application process is not suspended because of a call to KEYPOSITION.
- Error if Incomplete Nowait Operations Pending

  A call to the KEYPOSITION procedure is rejected with an error indication if there are any incomplete nowait operations pending on the specified file.
- Positioning on Duplicate or Nonexistent Records

  No searching of indexes is done by KEYPOSITION; therefore, a nonexistent or duplicate record is not reported until a subsequent READ, READUPDATE, WRITEUPDATE, LOCKREC, READLOCK, READUPDATELOCK, or WRITEUPDATEUNLOCK is performed.
- KEYPOSITION and Disc Seeks

  KEYPOSITION does not cause the disc heads to be repositioned; the heads are repositioned when a subsequent I/O call (READ, READUPDATE, WRITE, and so forth) transfers data.

- Positioning Exact

  If an exact KEYPOSITION is performed, and a <compare-length> is specified which is less than that specified when the file was created, <compare-length> must match the variable key length specified when the record is entered into the file. Otherwise, a subsequent call to READ, READUPDATE, WRITEUPDATE, LOCKREC, READLOCK, READUPDATELOCK, or WRITEUPDATEUNLOCK is rejected.

- Current-State Indicators After a KEYPOSITION

  Current-state indicators following a successful KEYPOSITION are:

current position	is that of the record indicated by the <key-value>, <key-specifier>, <positioning-mode>, and <compare-length> or the subsequent record if <positioning-mode>.<0> is set to 1.
positioning mode	is <positioning-mode> if the parameter is supplied; otherwise, it is an approximate value.
compare length	is <compare-length> if the <length-word> parameter is supplied; otherwise, it is the defined length of the specified key field.

  The compare length for generic searches is determined as follows:

```
IF <length-word>.<0:7> <> 0
 THEN <length-word>.<0:7>
  ELSE
   IF <length-word>.<8:15> > length of <key-specifier>
    THEN length of <key-specifier>
     ELSE <length-word><8:15>

   ! current primary key is <key-value> if <key-specifier> is
   ! primary, otherwise unchanged.
```

- Saving Current Position for Later Access

  When processing by alternate key, saving the concatenated alternate-key value and primary key values in a temporary buffer permits you to return to that position in a key-sequenced file; for example:

```
<temporary-buffer> ':=' (2 bytes for the altkey tag,
          record.altkey field for $LEN (record.altkey field), and
          record.primary key for $LEN (record.primary key);
```

Use the following to reposition to the same record:

```
KEYPOSITION ( <filenum> , <temporary-buffer>
            , <key-specifier> , <compare-length for generic
            searches '<<' 8 + length of alternate key + length
            of primary key>, <positioning-mode> );
```

Use the following to reposition to the next record:

```
KEYPOSITION ( <filenum> , <temporary-buffer>
            , <key-specifier> , <compare-length for generic
             searches '<<' 8 + length of alternate key +
             length of primary key> , <%100000 +
             positioning-mode> );
```

The <key-specifier> specifies the alternate key.

- Positioning to the Start of a File

  To position to the first record of a file, you can use the following call to specify a zero <length-word>:

```
  .
INT ZERO := 0;
CALL KEYPOSITION ( FILENUM , ZERO ,, ZERO );
```

- Maximum Size of a Relative File

  KEYPOSITION cannot be used on a relative file with more than 8,388,608 blocks (this is the approximate number that can be passed and stored in the 32-bit integer <record-specifier> parameter). Attempting to do so returns file system error 23 (illegal disc address).

- Positioning to the Middle of a Duplicate Alternate Key

  The normal case of positioning to the middle of a duplicate alternate key is to specify a value, in bits <8:15>, indicating the position within the file. You always get the first entry that matches the key for the supplied length. If you want to reposition by an alternate key, you must supply the <compare-length> in bits <0:7> and the length of the alternate key plus the primary key in bits <8:15>.

  The supplied <key-length> might not be valid, but this way the file system detects that the user wants to position by an alternate key behind the first entry which matches the supplied alternate key for <compare-length>.

- KEYPOSITION and File System Error 21

  If any of the following conditions are true, error 21 is returned by KEYPOSITION:

  - If the primary file is a key-sequenced file and one of the following is true:

    --<key-specifier> is omitted or 0 and <key-length> is greater than the key length defined for the primary file.

    --<compare-length> is greater than <key-length>.

  - If the <key-specifier> is not zero and one of the following is true:

    --<key-length> is greater than the sum of length of the alternate-key field and the length of the primary key of the file.

    --<key-length> is less than or equal to the length of the alternate-key field, and <compare-length> is greater than <key-length>.

    --<key-length> is greater than the length of the alternate-key field and the primary file is not key-sequenced, and the difference of <key-length> and <compare-length> is less than 4.

    --<key-length> is greater than the length of the alternate-key field and the primary file is not key-sequenced, and the <key-length> is less than the sum of the length of the alternate-key field and the length of the primary key of the file.

Example

```
KEY ':=' "BROWN";
COMPARE^LEN := 5;

CALL KEYPOSITION ( INFILE , KEY , , COMPARE^LEN );
```

Related Programming Manual

For programming information about the KEYPOSITION file system procedure, refer to the ENSCRIBE Programming Manual.

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.

The syntax for LASTADDR is:

```
<last-addr> := LASTADDR;

<last-addr>                    returned value

  INT

  returns the 'G'[0] relative word address of the last word in
  the application process's data area.
```

Condition Code Settings

The condition code has no meaning following a call to LASTADDR.

Example

```
NUM^PAGES := LASTADDR.<0:5> + 1;
```

The above function is used to determine the number of memory pages allocated to a running application program. A bit extraction is performed on the six high-order address bits returned from LASTADDR which is the page number. (One is added because pages are numbered from 0/n -1, but the number of pages is n.)

Related Programming Manual

For information about the LASTADDR utility procedure, refer to the GUARDIAN Operating System Programmer's Guide.

LASTRECEIVE PROCEDURE

The LASTRECEIVE procedure is used to obtain the process ID (PID) and the message tag associated with the last message read from the $RECEIVE file. This information is contained in the file's main-memory resident access control block (ACB). An application process is not suspended because of a call to LASTRECEIVE.

NOTE

To avoid receiving invalid information stored in the $RECEIVE part of the ACB, call the LASTRECEIVE procedure immediately following the call to READUPDATE for $RECEIVE (or the AWAITIO that completes the READUPDATE). Do not perform another READUPDATE of $RECEIVE before calling LASTRECEIVE. However, you can check the condition code or call FILEINFO without changing the information in the ACB.

The syntax for LASTRECEIVE is:

```
CALL LASTRECEIVE ( [ <process-id> ]                    ! o
                  ,[ <message-tag> ] );                ! o
```

<process-id> output

INT:ref:4

returns the ID of the process that sent the last message read through the $RECEIVE file. If the process is in the destination control table (DCT), the information returned consists of:

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

If the process is not in the DCT, the information returned consists of:

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

→

<message-tag> output

INT:ref:1

is used when the application process performs message queuing. <message-tag> returns a value that identifies the request message just read among other requests currently queued. To associate a reply with a given request, <message-tag> is passed in a parameter to the REPLY procedure.

The value of <message-tag> is the lowest integer between 0 and <receive-depth> minus 1, inclusive, that is not currently being used as a message tag. When a reply is made, its associated message tag value is made available for use as a message tag for a subsequent request message.

Condition Code Settings

< (CCL) indicates that $RECEIVE is not open.

= (CCE) indicates that LASTRECEIVE was successful.

> (CCG) does not return from LASTRECEIVE.

Considerations

- The PID That Is Returned by LASTRECEIVE

  The PID returned by LASTRECEIVE following receipt of a preceding OPEN, CLOSE, CONTROL, SETMODE, RESETSYNC, or CONTROLBUF system message identifies the process associated with the operation.

- High-Order Three Words of the <process id>

  The high-order three words of the PID will be 0 following the receipt of system messages other than OPEN, CLOSE, CONTROL, SETMODE, RESETSYNC, and CONTROLBUF.

Example

```
CALL LASTRECEIVE ( PROG1^ID );
```

The LASTRECEIVE procedure returns the identification of the process that sent the last message.

Related Programming Manual

For programming information about the LASTRECEIVE file system procedure, refer to the GUARDIAN Operating System Programmer's Guide.

LOCATESYSTEM PROCEDURE

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

The syntax for LOCATESYSTEM is:

```
{ <ldev> := } LOCATESYSTEM ( <sysnum>                    ! i, o
{ CALL      }                ,[ <sysname>] );            ! i
```

<ldev> — returned value

INT:value

returns one of the following values:

- -1 = all paths to the specified system are down.
- 0 = the system is undefined.
- >0 = is the logical device number of the line handler in the specified system.

<sysnum> — input, output

INT:ref:1

is the number of the system to be located unless you specify <sysname>. If you specify <sysname>, then the system number that corresponds to <sysname> returns into <sysnum>.

<sysname> — input

INT:ref:4

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

Condition Code Settings

The condition code has no meaning following a call to LOCATESYSTEM.

Considerations

- If the caller provides <sysname>, <sysnum> is returned the corresponding number, but if the caller omits <sysname>, the caller must supply <sysnum>.
- If the <sysname> specified does not exist, <sysnum> is set to 255.

Example

```
LDEV := LOCATESYSTEM ( SYS^NUM , SYS^NAME );
```

Related Programming Manual

None

LOCKFILE PROCEDURE

The LOCKFILE procedure is used to exclude other processes from accessing a file (and any records within that file).

If the file is currently unlocked or is locked by the caller when LOCKFILE is called, the file (and all its records) becomes locked, and the caller continues executing.

There are two "locking" modes available if the file is already locked:

- Default--Process requesting lock is suspended (see "Considerations").
- Alternate--Lock request is rejected (see "Considerations").

NOTE

Process suspension due to a queued lock occurs when the file is locked in the alternate locking mode and AWAITIO is called. AWAITIO returns the error that the "file is locked."

The syntax for LOCKFILE is:

```
CALL LOCKFILE ( <filenum>                          ! i
              ,[ <tag> ]  );                       ! o
```

<filenum> input

INT:value

is the number of an open file that identifies the file to be locked.

<tag> output

INT(32):value

is for nowait I/O only. <tag> is a value you define that uniquely identifies the operation associated with this LOCKFILE.

→

NOTE

The system stores the <tag> value until the I/O operation completes. The system then returns the <tag> information to the program in the <tag> parameter of the call to AWAITIO, thus indicating that the operation completed.

Condition Code Settings

< (CCL) indicates that an error occurred (call FILEINFO).

= (CCE) indicates that the LOCKFILE was successful.

> (CCG) indicates that the file is not a disc file.

Considerations

- Nowait and LOCKFILE

  If the LOCKFILE procedure is used to initiate an operation with a file-opened nowait, it must complete with a corresponding call to the AWAITIO procedure.

- Locking Modes

  --Default Mode

  If the file is already locked when the call to LOCKFILE is made, the process requesting the lock is suspended and queued in a "locking" queue behind other processes trying to access the file. When the file becomes unlocked, the process at the head of the locking queue is granted access to the file. If the process at the head of the locking queue is requesting a lock, it is granted the lock and resumes execution. If the process at the head of the locking queue is requesting a READ, the READ operation continues to completion.

  --Alternate Mode

  If the file is already locked when the call to LOCKFILE is made, the lock request is rejected, and the call to LOCKFILE completes immediately with error 73 ("file is locked"). The alternate locking mode is specified by calling the SETMODE procedure and specifying function 4.

- Locks and Open Files

  Locks are granted on an open file (that is, file number) basis. Therefore, if a process has multiple OPENs of the same file, a lock of one file number excludes access to the file through other file numbers.

- Attempting to READ a Locked File in Default Locking Mode

  If the default locking mode is in effect when a call to READ or READUPDATE is made to a locked file but the <filenum> of the locked file and the <filenum> supplied in the call differ, the caller of READ or READUPDATE is suspended and queued in the "locking" queue behind other processes attempting to access the file.

  NOTE

  A deadlock condition--a permanent suspension of your application-- occurs if READ or READUPDATE is called by the process which has a record locked by a <filenum> other than that supplied in READ or READUPDATE. In this case, the processes wait to resolve the deadlock until someone stops one of the processes. (Refer to the OPEN procedure for an explanation of multiple OPENs by the same process.)

- Accessing a Locked File

  If the <filenum> of the locked file and the <filenum> in the call differ, the call is rejected with file system error 73 ("file is locked") when:

  --READ or READUPDATE is called, and the alternate locking mode is in effect.

  --WRITE, WRITEUPDATE, or CONTROL is called.

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

- Locks Are not Nested

  For example:

```
.
CALL LOCKFILE ( FILE^A,...);     ! FILE^A becomes locked.
.
CALL LOCKFILE ( FILE^A,...);     ! is a null operation, because
                                 ! the file is already locked.
                                 ! A condition code of CCE
                                 ! returns.
.
CALL UNLOCKFILE ( FILE^A,...);   ! FILE^A becomes unlocked.
.
CALL UNLOCKFILE ( FILE^A,...);   ! is a null operation, because
                                 ! the file is already unlocked.
                                 ! A condition code of CCE
                                 ! returns.
```

Example

```
CALL LOCKFILE ( FILE^NUM );
```

Related Programming Manual

For programming information about the LOCKFILE file system procedure, refer to the ENSCRIBE Programming Manual and the GUARDIAN Operating System Programmer's Guide.

LOCKREC PROCEDURE

The LOCKREC procedure excludes other processes from accessing a record at the current position. For key-sequenced, relative, and entry-sequenced files, the current position is the record with a key value that matches exactly the current key value. For unstructured files, the current position is the relative byte address (RBA) identified by the current-record pointer.

If the record is unlocked when LOCKREC is called, the record becomes locked, and the caller continues executing.

There are two locking modes available if the record is already locked:

- Default--Process requesting lock is suspended (see "Considerations").
- Alternate--Lock request is rejected (see "Considerations").

NOTE

A call to LOCKREC is not equivalent to locking all records in a file; that is, locking all records still allows insertion of new records, but file locking does not. File locks and record locks are queued in the order they are issued.

The syntax for LOCKREC is:

```
CALL LOCKREC ( <filenum>                                  ! i
             ,[ <tag> ] );                                ! i
```

<filenum> input

INT:value

is the number of an open file that identifies the file containing the record to be locked.

<tag> input

INT(32):value

is for nowait I/O only. <tag> is a value you define that uniquely identifies the operation associated with this LOCKREC.

→

NOTE

The system stores the <tag> value until the I/O operation completes. The system then returns the <tag> information to the program in the <tag> parameter of the call to AWAITIO, thus indicating that the operation completed.

Condition Code Settings

< (CCL) indicates that an error occurred (call FILEINFO).

= (CCE) indicates that the LOCKREC was successful.

> (CCG) indicates that the file is not a disc file.

General Considerations

- Nowait and LOCKREC

  If the LOCKREC procedure is used to initiate an operation with a file-opened nowait, it must complete with a corresponding call to the AWAITIO procedure.

  Process suspension due to a queued lock occurs when the file is locked in the alternate locking mode and AWAITIO is called. AWAITIO returns error "file is locked."

- Default Locking Mode

  If the record is already locked when LOCKREC is called, the process requesting the lock is suspended and queued in a "locking" queue behind other processes also requesting to lock or read the record.

  When the record becomes unlocked, the process at the head of the locking queue is granted access to the record. If the process at the head of the locking queue is requesting a lock, it is granted the lock and resumes execution. If the process at the head of the locking queue is requesting a READ operation, the READ operation continues to completion.

- Alternate Locking Mode

  If the record is already locked when LOCKREC is called, the lock request is rejected, and the call to LOCKREC completes immediately with file system error 73 ("record is locked"). The alternate locking mode is specified by calling the SETMODE procedure and specifying function 4.

- Accessing a Locked File

  If the <filenum> of the locked file and the <filenum> in the call differ, the call is rejected with file system error 73 ("file is locked") when:

  --READ or READUPDATE is called, and the alternate locking mode is in effect.

  --WRITE, WRITEUPDATE, or CONTROL is called.

- Attempting to Read a Locked Record in Default Locking Mode

  If the default locking mode is in effect when a READ or READUPDATE is called for a record that is locked, and the <filenum> of the file containing the record differs from the <filenum> in the call, the caller to READ or READUPDATE is suspended and queued in the "locking" queue behind other processes attempting to lock or read the record.

  NOTE

  A deadlock condition--a permanent suspension of your application-- occurs if READ or READUPDATE is called by the process which has a record locked by a <filenum> other than that supplied to READ or READUPDATE. In this case, the processes wait to resolve the deadlock until someone stops one of the processes. (Refer to the OPEN procedure for an explanation of multiple OPENs by the same process.)

- Selecting the Locking Mode With SETMODE

  The locking mode is specified by the SETMODE procedure with <function> = 4.

- Locks Cannot Be Nested

  Locks are not nested; for example:

```
CALL LOCKREC ( file^a,... ); ! locks the current record in
                             ! "file^a."
  ...
CALL LOCKREC ( file^a,... ); ! has no effect since the
                             ! current record is already
                             ! locked.
  ...
CALL UNLOCKREC (file^a,...); ! unlocks the current record
                             ! in "file^a."
  ...
CALL UNLOCKREC (file^a,...); ! has no effect since the
                             ! current record is not locked.
```

Disc File Considerations

- Structured Files

  --Calling LOCKREC after positioning on a nonunique key

  If the call to LOCKREC immediately follows a call to KEYPOSITION where a nonunique alternate key is specified, the LOCKREC fails. A subsequent call to FILEINFO returns file system error 46 (invalid key). However, if an intermediate call to READ is performed, the call to LOCKREC is permitted because a unique record is identified.

  --Current-state indicators after LOCKREC

  After a successful LOCKREC, current-state indicators are unchanged.

- Unstructured Files

  --Locking the RBA in an unstructured file

  Record positions in an unstructured file are represented by an RBA, and the RBA can be locked with LOCKREC. To lock a position in an unstructured file, first call POSITION with the desired RBA, and then call LOCKREC. This locks the RBA; any other process attempting to access the file with exactly the same RBA encounters a "record is locked condition." You can access that RBA by positioning to RBA-2. Depending on the process's locking mode, the call either fails with file system error 73 ("record is locked") or is placed in the locking queue.

  --Record pointers after LOCKREC

  After a call to LOCKREC, the current-record, next-record, and end-of-file pointers remain unchanged.

- Deadlock

  --Ways to avoid deadlocks

  One way to avoid deadlock is to use the alternate locking mode, established by <function> 4 of the SETMODE procedure. Another common method of avoiding deadlock situations is to lock and unlock records in some predetermined order.

Example

```
CALL LOCKREC ( FILE^NUM , LOCK^TAG );
```

Related Programming Manual

For programming information about the LOCKREC file system procedure, refer to the ENSCRIBE Programming Manual.

LOOKUPPROCESSNAME PROCEDURE

The LOOKUPPROCESSNAME procedure is used to obtain a description of a named process pair by its name or by its index into the local destination control table (DCT). To obtain remote process pair descriptions by index, use the GETPPDENTRY procedure.

The syntax for LOOKUPPROCESSNAME is:

```
CALL LOOKUPPROCESSNAME ( <ppd> );                    ! i, o

<ppd>                          input, output

  INT:ref:9

  on input, is either:

  --the process name
  --the entry number in the DCT ({0:n})

  for the entry to be returned.

  On return, <ppd> is of the form:

    <ppd>[0:2]      = process name of entry

        [3].<0:7>   = <cpu> for primary process
           .<8:15>  = <pin> for primary process

        [4].<0:7>   = <cpu> of backup process, else 0
           .<8:15>  = <pin> of backup process, else 0

        [5:8]       = <process-id> of ancestor

  If the process name is not in the DCT, <ppd> is
  unchanged.
```

Condition Code Settings

< (CCL) indicates that the specified process name is not in the directory, or that the remote system could not be accessed.

= (CCE) indicates that the specified name was found.

> (CCG) indicates that the specified entry number exceeds the last table entry.

Considerations

- Network Use

  Remote DCT entries can be obtained by passing the process name (in network form) of the process desired. On return, the process name remains in network form.

  This is an example of using LOOKUPROCESSNAME to get the DCT entry for the name process "$PROC" running on the system "\DETROIT":

```
EXTERNAL^NAME ':=' 17 * [ " " ];  ! blanks.
EXTERNAL^NAME ':=' "\DETROIT.$PROC";
CALL FNAMEEXPAND ( EXTERNAL^NAME , INTERNAL^NAME , DEFAULTS );
   ! converts \DETROIT to its system number.
CALL LOOKUPPROCESSNAME ( INTERNAL^NAME );
   ! returns the desired DCT entry.
```

  To obtain DCT entries using an <entry-num>, use the GETPPDENTRY procedure.

- Scanning the Entire DCT

  If the <ppd-entry> you specify is not currently being used, LOOKUPPROCESSNAME returns CCE and sets <ppd-entry> to zeros. By using repeated entry numbers, the entire DCT can be scanned.

  The LOOKUPROCESSNAME always returns CCE until the <ppd-entry> is greater than the number of entries in the DCT.

Example

```
CALL LOOKUPPROCESSNAME ( ENTRY );
```

Related Programming Manual

For programming information about the LOOKUPPROCESSNAME process control procedure, refer to the GUARDIAN Operating System Programmer's Guide.

MOM PROCEDURE

The MOM procedure provides a process with the process ID (PID) of its creator.

The syntax for MOM is:

```
CALL MOM ( <process-id> );                          ! o

<process-id>                  output

  INT:ref:4

  is an array of four words where MOM returns the PID of the
  caller's creator.  For an nonnamed process, <process-id> is:

    <process-id>[0].<0:1>   = 2
                   .<2:15>  = word 0 (creation timestamp)

                              where:

                              .<2:7>   = unused
                              .<8:15>  = system number (if the system
                                         is part of a network)
                                       = 0 otherwise

                  [1:2]      = words 1 and 2 (creation timestamp)

                  [3].<0:3>  = unused
                     .<4:7>  = <cpu> number where the process is
                               executing
                     .<8:15> = <pin> assigned by the operating system
                               to identify the process in the CPU

  For a named local process, <process-id> is:

    <process-id>[0:2]      = $<process-name>

                  [3]      = two blanks
                                or
                  [3].<0:7>  = <cpu> number
                     .<8:15> = <pin>
```

Condition Code Settings

The condition code has no meaning following a call to MOM.

Considerations

- Calling MOM From a Process Pair

  If the caller of MOM is the primary process of a named process pair and no backup process has been created, zeros are returned in <process-id>.

- When the PID of the Backup or the Primary Is Returned

  If the caller of MOM is the primary process of a named process pair and there is a backup process, the PID of the backup is returned.

  If the caller of MOM is the backup process of a named process pair, the PID of the primary is returned.

- Passing the PID to the System Procedures

  The PID returned from MOM is suitable for passing directly to any file system procedure. (If you pad the PID with blanks before or after the call to MOM, you can pass the PID as a file name to any system procedure.)

- How Calling the STEPMOM Procedure Affects the PID

  If another process has made itself the creator of the caller of MOM (through a call to STEPMOM), then the PID of that process is returned.

- Network Consideration

  If a process's creator is in a remote system, its PID is returned by MOM in network form. A process can use this fact to determine whether or not it is created locally.

Example

```
CALL MOM ( MY^CREATOR );
```

Related Programming Manual

For programming information about the MOM process control procedure, refer to the GUARDIAN Operating System Programmer's Guide.

MONITORCPUS PROCEDURE

The MONITORCPUS procedure instructs the GUARDIAN operating system to notify the application process if a designated processor module either:

- Fails (indicated by the absence of an operating system "I'm alive" message)
- Returns from a failed to an operable state (that is, reloaded by means of a command interpreter RELOAD command)

The calling application process is notified by means of a system message read through the $RECEIVE file.

The syntax for MONITORCPUS is:

```
CALL MONITORCPUS ( <cpu-mask> );        ! i

<cpu-mask>                      input

  INT:value

  is a bit that is set to "1," corresponding to each processor
  module to be monitored:

    <cpu mask>.<0> = 1  processor module 0 to be monitored
              .<1> = 1  processor module 1 to be monitored
          .
          .
          .
    <cpu mask>.<14> = 1  processor module 14 to be monitored
    <cpu mask>.<15> = 1  processor module 15 to be monitored

    <cpu mask> = 0 means no notification occurs.
```

Condition Code Settings

The condition code has no meaning following a call to MONITORCPUS.

Messages

- CPU Down

  The CPU down message (-2) is received if failure occurs with a processor module that is being monitored (refer to Appendix F for the description and form of system messages).

- CPU Up

  This message is received if a reload occurs with a processor module that is being monitored (system message -3).

Refer to Appendix F for a list of system messages sent to processes.

Example

```
CALL MONITORCPUS( %100000 '>>' BACKUP^CPU );    ! monitor the
                                                ! backup CPU.
```

Related Programming Manual

For programming information about the MONITORCPUS checkpointing procedure, refer to the GUARDIAN Operating System Programmer's Guide.

MONITORNET PROCEDURE

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

The syntax for MONITORNET is:

```
CALL MONITORNET ( <enable> );          ! i

<enable>                      input

  INT:value

  contains one of the following values:

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

Condition Code Settings

The condition code has no meaning following a call to MONITORNET.

Considerations

- To Receive Status Changes for Local Processors

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

Message

- Change in Status of Network Processors

  A process that has enabled MONITORNET receives a system message (-8) through $RECEIVE whenever a change in the status of a remote processor occurs. The processor status bit masks have a 1 in bit <cpu number> to indicate that the processor is up and a 0 to indicate that the processor is down. (See Appendix F for a list of system messages sent to processes.)

Example

```
CALL MONITORNET ( 1 );
```

Related Programming Manual

None

MONITORNEW PROCEDURE

The MONITORNEW procedure enables or disables receipt of new system messages, including the SETTIME and Power On messages.

The syntax for MONITORNEW is:

```
CALL MONITORNEW ( <enable> );          ! i

<enable>                    input

  INT:value

  contains one of the following values:

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

Condition Code Settings

The condition code has no meaning following a call to MONITORNEW.

Considerations

- When to Enable or Disable Receipt of Messages

  The SETTIME and Power On messages are not received unless the process makes a call to MONITORNEW with <enable> set to 1. To disable receipt of these messages, the process must make another call, setting <enable> to 0.

Example

```
CALL MONITORNEW ( 1 );
```

Related Programming Manual

For programming information about the MONITORNEW file system procedure, refer to the GUARDIAN Operating System Programmer's Guide.

MYPID PROCEDURE

The MYPID procedure provides a process with its own CPU and pin number.

The syntax for MYPID is:

```
<cpu,pin> := MYPID;
```

<cpu,pin>	returned value
INT:value	is the caller's processor (bits <0:7>) and pin number (bits <8:15>).

Condition Code Settings

The condition code has no meaning following a call to MYPID.

Example

```
ME := MYPID;
```

Related Programming Manual

For programming information about the MYPID process control procedure, refer to the GUARDIAN Operating System Programmer's Guide.

MYPROCESSTIME PROCEDURE

The MYPROCESSTIME procedure returns the process execution time of the calling process (in microseconds).

The syntax for MYPROCESSTIME is:

```
<process-time> := MYPROCESSTIME;
```

<process-time>	returned value
FIXED	is a value representing the clock of the current process in microseconds.

Condition Code Settings

The condition code has no meaning following a call to MYPROCESSTIME.

Example

```
RET^CLOCK := MYPROCESSTIME;
```

Related Programming Manual

None

MYSYSTEMNUMBER PROCEDURE

The MYSYSTEMNUMBER procedure provides a process with its own system number.

The syntax for MYSYSTEMNUMBER is:

```
<sysnum> := MYSYSTEMNUMBER;

<sysnum>                        returned value

  INT

  is the caller's system number.
```

Condition Code Settings

The condition code has no meaning following a call to MYSYSTEMNUMBER.

Considerations

- Part of Network or Local System

  The following IF (skeleton) statement determines if you are running on a network system.

```
IF NOT ( SYS^NUM := MYSYSTEMNUMBER ) THEN
   ! not on network system
```

If the caller is running in a local nonnamed system, MYSYSTEMNUMBER returns 0. Since 0 is a legal system number, a process wishing to determine the name of the system on which it is running can use the following call.

```
CALL GETSYSTEMNAME( MYSYSTEMNUMBER, NAME );
```

A return of all blanks in a name indicates that the system is not part of a network, or that it is a local system which is not named.

Example

See "Considerations" above.

Related Programming Manual

None

MYTERM PROCEDURE

The MYTERM procedure provides a process with the file name of its home terminal. The file name returned from MYTERM is suitable for passing directly to any file system procedure that accepts a file name in internal form.

The syntax for MYTERM is:

```
CALL MYTERM ( <filename> );                          ! o

<filename>                    output

  INT:ref:12

  is an array of 12 words where MYTERM returns the device name
  and the subdevice name, if any, of the home terminal in one of
  the following two forms:

    $<devname>[#<subdev-name>]
    $<process name>[#<subname>]
```

Condition Code Settings

The condition code has no meaning following a call to MYTERM.

Considerations

- File Name Form

  The file name returned from MYTERM is the same form as that used by the file system procedures.

- How the Home Terminal Is Determined

  The home terminal is always the same as the home terminal of a process's true creator (not STEPMOM), unless the home terminal is altered by SETMYTERM or the <hometerm> parameter is supplied in the NEWPROCESS procedure.

  If the process calling MYTERM is a descendant of a command interpreter, then the home terminal is the same as that of the command interpreter or that of an explicit TERM specifier on the RUN command.

Example

```
CALL MYTERM ( HOME^TERM );  ! returns the name of the home
                            ! terminal.
```

Related Programming Manual

For programming information about the MYTERM process control procedure, refer to the GUARDIAN Operating System Programmer's Guide.

NEWPROCESS PROCEDURE

The NEWPROCESS procedure is used to create and, optionally, to assign a symbolic process name to a new process. Additionally, you can specify the execution priority of the new process, the number of memory pages allotted the process, and the processor where the process is to execute. You can also specify a user library and a swap file. When a new process is created, its process ID (PID) is returned to the caller.

The syntax for NEWPROCESS is:

```
CALL NEWPROCESS ( <filenames>                                  ! i
                 ,[ <priority> ]                               ! i
                 ,[ <memory-pages> ]                           ! i
                 ,[ <processor> ]                              ! i
                 ,[ <process-id> ]                             ! o
                 ,[ <error> ]                                  ! o
                 ,[ <name> ]                                   ! i
                 ,[ <hometerm> ]                               ! i
                 ,[ <inspect-flag> ]   );                      ! i
```

<filenames> input

INT:ref:12 or INT:ref:36

is an array that contains the 12-word file name of the program to be run and, optionally, two additional fields. Refer to the GUARDIAN Operating System Programmer's Guide for information about file names.

The additional fields which are used only if bit 1 of the <priority> parameter is set to 1, are as follows:

<filename>.[12:23] = <library-file>

is the 12-word file name of a user library to be used by the program. (The user library must be on the same node as the process that was created.)

→

<filename>.[24:35] = <swap-file>

is the 12-word file name of a file to be used as a swap file for the data stack. (The swap file must be on the same node as the process that was created.)

See "Considerations" for more information.

<priority> input

INT:value

is a value consisting of three parts:

<priority>.<0> is the DEBUG bit. If <priority>.<0> = 1, then a code breakpoint is set on the first executable instruction of the program's MAIN procedure.

.<1> indicates the interpretation of the additional fields of the <filenames> parameter. If <priority>.<1> = 1, the additional fields in <filenames> are used. If <priority>.<1> = 0, these extra fields are ignored.

.<2:7> should be 0.

.<8:15> is the execution priority to be assigned to the new process {1:199}. If <priority>.<8:15> = 0, then the priority of the caller of NEWPROCESS is used. If a value greater than 199 is specified, then 199 is used.

If <priority> is omitted, the caller's priority is used.

<memory-pages> input

INT:value

specifies the number of 1024-word memory pages allotted to the new process. If <memory-pages> is omitted or is less than the value assigned when the program is compiled (or created with BINDER), then the compilation value is used. In any case, the maximum number of pages permitted is 64.

<processor> input

INT:value

specifies the processor where the new process runs. If omitted, the new process runs in the same processor as the caller.

<process-id> output

INT:ref:4

is a four-word array where NEWPROCESS returns the PID of the new process. If the new process was created in:

--The local system, the local form of the PID is returned.
--The remote system, the network form of the PID is returned. (A new process is created on a remote system on the same node where its program file resides.)

If no process was created, zero is returned into <process id>.

<error> output

INT:ref:1

returns a number indicating the outcome of the process creation attempt, where:

<error>.<0:7>

0 = no error, process created
1 = warning that the process had undefined externals, but was started
2 = no process control block available
3 = file system error occurred on program file, then
<error>.<8:15> = file system error number
4 = unable to allocate map
5 = file error on swap file , then
<error>.<8:15> = file system error number
6 = illegal file format, then
<error>.<8:15> = 0, program file is in error
= 1, library file is in error
7 = unlicensed privileged program
8 = process name error, then
<error>.<8:15> = file system error number
9 = library conflict

10 = unable to communicate with system monitor process
11 = file system error occurred on library file, then
<error>.<8:15> = file system error number
12 = program file and library file specified are same file
13 = extended data segment initialization error, then
<8:15> = file system error number
14 = extended segment swap file error, then <8:15> =
file system error number
15 = illegal home terminal, then
<error>.<8:15> = file system error number

Refer to the System Messages Manual for a list of all file system errors.

<name> input

INT:ref:3

if present, is a name to be given to the new process. It is entered into the destination control table. <name> is of the form:

<name>[0:2] = $<process-name>

<process-name> must be preceded by a dollar sign ("$") and consists of a maximum of five alphanumeric characters; the first character must be alphabetic. (If the process is created in a remote system and it is necessary to be able to access the process, its name should consist of, at most, four characters and the "$.")

<hometerm> input

INT:ref:12

is the home terminal for the new process. The default is the home terminal of the caller.

→

```
<inspect-flag>                input

  INT:value

  sets the debugging attributes for the new process:

    <flags>.<14> = 1    saveabend file creation
                 = 0    no saveabend file creation

    <flags>.<15> = 1    INSPECT
                 = 0    DEBUG

    When <flags> is specified, the bits <14> and <15> are
    ORed with the corresponding flags in the object code file.
    If <flags>.<14> is set but <flag>.<15> is not, then
    <flags>.<15> is also set.

    If these <flags> are omitted then the defaults are set from
    the flags in the object code file (set by compiler
    directives at compile time, after the object flags are ORed
    with the callers debugging attributes).
```

Condition Code Settings

The condition code has no meaning following a call to NEWPROCESS.

Considerations

- When BIT 1 of <priority> Is Set to 1

  To specify only one of the two extra fields, the calling process must set <priority>.<1> to 1 and fill the <filename> not specified with blanks.

  If <library-file>:

  a. is specified, unresolved external references are resolved first from the specified <library-file>, then from the system library.

  b. is specified and <library-file>.[0] is 0 (binary), then the library file used by the process when it was last run is removed, and the process runs with no library file. (The references that were previously resolved on the user library are resolved on the system library.)

10 = unable to communicate with system monitor process
11 = file system error occurred on library file, then
<error>.<8:15> = file system error number
12 = program file and library file specified are same file
13 = extended data segment initialization error, then
<8:15> = file system error number
14 = extended segment swap file error, then <8:15> =
file system error number
15 = illegal home terminal, then
<error>.<8:15> = file system error number

Refer to the System Messages Manual for a list of all file system errors.

<name> input

INT:ref:3

if present, is a name to be given to the new process. It is entered into the destination control table. <name> is of the form:

<name>[0:2] = $<process-name>

<process-name> must be preceded by a dollar sign ("$") and consists of a maximum of five alphanumeric characters; the first character must be alphabetic. (If the process is created in a remote system and it is necessary to be able to access the process, its name should consist of, at most, four characters and the "$.")

<hometerm> input

INT:ref:12

is the home terminal for the new process. The default is the home terminal of the caller.

→

```
<inspect-flag>                input

  INT:value

  sets the debugging attributes for the new process:

    <flags>.<14> = 1   saveabend file creation
                 = 0   no saveabend file creation

    <flags>.<15> = 1   INSPECT
                 = 0   DEBUG

    When <flags> is specified, the bits <14> and <15> are
    ORed with the corresponding flags in the object code file.
    If <flags>.<14> is set but <flag>.<15> is not, then
    <flags>.<15> is also set.

    If these <flags> are omitted then the defaults are set from
    the flags in the object code file (set by compiler
    directives at compile time, after the object flags are ORed
    with the callers debugging attributes).
```

Condition Code Settings

The condition code has no meaning following a call to NEWPROCESS.

Considerations

- When BIT 1 of <priority> Is Set to 1

  To specify only one of the two extra fields, the calling process must set <priority>.<1> to 1 and fill the <filename> not specified with blanks.

  If <library-file>:

  a. is specified, unresolved external references are resolved first from the specified <library-file>, then from the system library.

  b. is specified and <library-file>.[0] is 0 (binary), then the library file used by the process when it was last run is removed, and the process runs with no library file. (The references that were previously resolved on the user library are resolved on the system library.)

c. is not specified, the program runs with the same library file as the last time it was run (or no file if that was how it was run) or with the library file currently executing. Refer to the BINDER Manual for more information about user libraries.

If <swap-file>:

a. is specified and a file of that name exists, that file is used for memory swaps of the user data stack during execution of the process; if no file of that name exists, a file of that name and of the necessary size is created and used for swaps.

b. is not specified, a temporary file is created on the disc where the program file resides.

c. specifies only the disc device name (filling the rest of the file name with blanks), a temporary file is created on the specified disc device.

- Creation of the Backup of a Named Process Pair

  If the backup of a named process pair is created, the backup process becomes the "creator" of the primary (that is, the caller to NEWPROCESS).

- PID and the OPEN Procedure

  The PID returned from NEWPROCESS is suitable for passing directly to any of the file system procedures that accept file names (if blank-filled on the right).

- Limitations to the <filenames> Parameter

  In a network containing both NonStop 1+ systems and NonStop systems, a calling process on a NonStop 1+ system cannot use the extra fields in <filenames>, even to create a process on a NonStop system. If specified, the extra fields are ignored.

- Program File and User Library File Differences

  A "user library" is an object file containing one or more procedures. The difference between a program file and a library file is that the library file cannot contain a main procedure. Undefined externals from a library are resolved only from the system library. A program file must contain a main procedure. Refer to the Binder User's Manual for additional information about user libraries.

- Library Conflict--NEWPROCESS Error

  The library file for a process can be shared by any number of processes. However, when a program file is shared by two or more processes, all processes must have the same user library configuration; that is, all processes sharing the program either have the same user library, or they have no user library. An error 9 ("library conflict") occurs when a copy of the running program runs with a different library configuration than was specified in the call to NEWPROCESS.

Example

```
CALL NEWPROCESS ( PFILE^NAME , , , , PID , ERROR );
```

Related Programming Manual

For programming information about the NEWPROCESS process control procedure, refer to the GUARDIAN Operating System Programmer's Guide.

NEWPROCESSNOWAIT PROCEDURE

The NEWPROCESSNOWAIT procedure is used to create a new process in a nowait manner and, optionally, to assign a symbolic process name to it. Additionally, you can specify the execution priority of the new process, the number of memory pages allotted the process, and the processor where the process is to execute. You can also specify a user library and a swap file. When a new process is created, its process ID (PID) returns to the caller by a system message on the caller's $RECEIVE file.

The syntax for NEWPROCESSNOWAIT is:

```
CALL NEWPROCESSNOWAIT ( <filenames>                          ! i
                       ,[ <priority> ]                       ! i
                       ,[ <memory-pages> ]                   ! i
                       ,[ <processor> ]                      ! i
                       ,[ <process-id> ]                     ! unused
                       ,[ <error> ]                          ! o
                       ,[ <name> ]                           ! i
                       ,[ <hometerm> ]                       ! i
                       ,[ <inspect-flags> ]  );              ! i
```

<filenames> input

INT:ref:12 or INT:ref:38

is an array that contains the 12-word file name of the program to be run and three additional fields: <library-file>, <swap-file>, and <tag>. Refer to the GUARDIAN Operating System Programmer's Guide for information about file names.

The additional fields, which are used only if bit 1 of the <priority> parameter is set to 1, are as follows:

<filenames>[12:23] = <library-file>

is the 12-word file name of a user library to be used by the program.

<filenames>[24:35] = <swap-file>

is the 12-word file name of a file to be used as a swap file.

→

<tag> is a 2-word value used to identify the completion message from the call to NEWPROCESSNOWAIT.

See "Considerations" for more information.

<priority> input

INT:value

is a value passed out of <priority> that consists of three parts:

<priority>.<0> is the DEBUG bit. If <priority>.<0> = 1, then a code breakpoint is set on the first executable instruction of the program's MAIN procedure.

.<1> indicates the interpretation of the additional fields of the <filenames> parameter. If <priority>.<1> = 1, the additional fields in <filenames> are used. If <priority>.<1> = 0, these extra fields are ignored.

.<8:15> is the execution priority assigned to the new process {1:199}. If <priority>.<8:15> = 0, then the priority of the caller of NEWPROCESSNOWAIT is used. If a value greater than 199 is specified, then 199 is used.

If <priority> is omitted, the caller's priority is used.

<memory-pages> input

INT:value

specifies the number of 1024-word memory pages to be allotted the new process. If <memory-pages> is omitted or is less than the value assigned when the program is compiled (or created with BINDER), then the compilation value is used. In any case, the maximum number of pages permitted is 64.

<processor> input

INT:value

is a value specifying the CPU where the new process runs. If omitted, the new process runs in the same processor as the caller.

<process-id> unused

INT:ref:4

is not used by NEWPROCESSNOWAIT.

<error> output

INT:ref:1

returns a number indicating the initial outcome of the process creation attempt. Only errors that prevented initiation of process creation are reported in this parameter; if process creation was initiated, any subsequent errors are reported in the completion message on $RECEIVE. The error numbers in the <error> parameter are:

<error>.<0:7>

```
  0 = no error, process creation initiated
  3 = file system error occurred on program file
      <error>.<8:15> = file system error number
  5 = file error
      <error>.<8:15> = file system error number
  8 = process name error
      <error>.<8:15> = file system error number
 10 = unable to communicate with system monitor process
 11 = file system error occurred on library file
      <error>.<8:15> = file system error number
 15 = illegal home terminal, then
      <error>.<8:15> = file system error number
```

Refer to the System Messages manual for a list of all NEWPROCESS errors.

→

<name> input

INT:ref:3

if present, is a name to be given to the new process. It is entered into the the destination control table. <name> is of the form:

<name>[0:2] = $<process-name>

<process-name> must be preceded by a dollar sign ("$") and consists of a maximum of five alphanumeric characters; the first character must be alphabetic. (If the process is created in a remote system and it is necessary to be able to access the process, its name should consist of, at most, four characters and the "$.")

<hometerm> input

INT:ref:12

is the home terminal for the new process. The default is the home terminal of the caller.

<inspect-flags> input

INT:value

sets the debugging attributes for the new process:

<flags>.<14> = 1 saveabend file creation
= 0 no saveabend file creation

<flags>.<15> = 1 INSPECT
= 0 DEBUG

When <flags> is specified, the bits <14> and <15> are ORed with the corresponding flags in the object code file. If <flags>.<14> is set but <flags>.<15> is not, then <flags>.<15> is also set.

If <flags> is omitted, then the defaults are set from the flags in the object code file (set by compiler directives at compile time).

Condition Code Settings

The condition code has no meaning following a call to NEWPROCESSNOWAIT.

Considerations

- When Bit 1 of <priority> Is Set to 1

  The value in the <tag> parameter appears in the message returned upon completion of NEWPROCESSNOWAIT. To specify only one or two of the three extra fields, the calling process must set <priority>.<1> to 1 and fill the field(s) not to be specified with blanks.

  If <library-file>:

  a. is specified, unresolved external references are resolved first from the specified <library-file>, then from the system library.

  b. is specified and <library-file>.[0] is 0, then the library file used by the process when it was last run is removed, and the process runs with no library file. (The references that were previously resolved on the user library are resolved on the system library.)

  c. is not specified, the program runs with same library file as the last time it was run (or no file if that was how it was run) or with the library file currently executing.

  If <swap-file>:

  a. is specified and a file of that name exists, that file is used for memory swaps of the user data stack during execution of the process; if no file of that name exists, a file of that name and of the necessary size is created and used for swaps.

  b. is not specified, a temporary file is created on the disc where the program file resides.

  c. specifies only the device name (filling the rest of the file name with blanks), a temporary file is created on the specified device.

- When a Nonzero Value Is Returned in <error>

  If NEWPROCESSNOWAIT cannot initiate process creation (for instance, if an invalid processor number is specified), no message appears on $RECEIVE. The <error> parameter is returned a nonzero value indicating the error.

- Using NEWPROCESSNOWAIT Remotely On the Correct System

  Do not attempt to remotely start a process on a NonStop 1+ system. using the NEWPROCESSNOWAIT procedure. The NonStop 1+ system does not support the NEWPROCESSNOWAIT procedure and will not send a new process completion message (-12) to the caller's $RECEIVE file. Use the NEWPROCESS procedure instead.

- See "Considerations" under the NEWPROCESS procedure.

Message

- NEWPROCESSNOWAIT Completion Message

  If NEWPROCESSNOWAIT succeeds in initiating process creation or an error occurs during process creation a system message -12 appears on $RECEIVE upon completion. (Refer to Appendix F for a complete description of system messages.)

Example

```
CALL NEWPROCESSNOWAIT ( PFILE^NAME
                      ,               ! priority.
                      ,               ! memory pages.
                      ,               ! processor.
                      ,               ! PID.
                      , ERROR
                      , NEW^NAME );
```

Related Programming Manual

For programming information about the NEWPROCESSNOWAIT process control procedure, refer to the GUARDIAN Operating System Programmer's Guide.

NEXTFILENAME PROCEDURE

The NEXTFILENAME procedure is used to obtain the name of the next disc file on a designated volume. NEXTFILENAME returns the next file name in alphabetic sequence after the file name supplied as the parameter. The alphabetic sequence includes digits 0-9; if the volume contains temporary files, the first temporary file is returned when <filename> is $<volname><blank-fill>.

The intended use of NEXTFILENAME is in an iterative loop, where the file name returned in one call to NEXTFILENAME specifies the starting point for the alphabetic search in the subsequent call to NEXTFILENAME. In this manner, a volume's file names are returned to the application process in alphabetic order through successive calls to NEXTFILENAME.

The syntax for NEXTFILENAME is:

```
<error> := NEXTFILENAME ( <filename> );      ! i, o
```

<error> returned value

INT

is a file system error number indicating the outcome of the call. Common errors returned are:

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

Refer to the System Messages Manual for a list of all file system errors.

<filename> input, output

INT:ref:12

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

→

To obtain the name of the first file on $<volname>:

```
<filename>[0:11] = $<volname><blank-fill>
                          or
                   \<sysnum><volname><blank-fill>
```

To obtain the name of the first file in <subvol-name> on $<volume>:

```
<filename>[0:3]  = $<volname><blank-fill>
                          or
                   \<sysnum><volname><blank-fill>

<filename>[4:11] = <subvol-name><blank-fill>
```

To return the name of the next file in alphabetic sequence:

```
<filename>[0:3]  = $<volname><blank-fill>
                          or
                   \<sysnum><volname><blank-fill>

<filename>[4:7]  = <subvol-name><blank-fill>
<filename>[8:11] = <disc-filename><blank-fill>
```

When <filename> returns, it contains the next file name, if any, in alphabetic sequence.

Condition Code Settings

The condition code has no meaning following a call to NEXTFILENAME.

Example

```
FNAME ':=' [ "$SYSTEM ", 8 * ["  "]];
WHILE NOT (ERROR := NEXTFILENAME ( FNAME ) ) DO
   BEGIN
      .
      .
      .
   END;
```

Related Programming Manual

For programming information about the NEXTFILENAME file system procedure, refer to the ENSCRIBE Programming Manual.

NUMIN PROCEDURE

The NUMIN procedure converts the ASCII characters used to represent a number into the signed integer value for that number.

The syntax for the NUMIN function is:

```
{ <next-addr> := } NUMIN ( <ascii-num>                ! i
{ CALL           }        ,<signed-result>            ! o
                          ,<base>                     ! i
                          ,<status> );                ! o
```

<next-addr> returned value

INT

returns the 'G'[0] relative string address of the first character in <ascii-num> not used in the conversion.

<ascii-num> input

STRING:ref:*

is an array containing the number to be converted to signed integer form. <ascii-num> is of the form:

```
[ + ] [ % ] <number> <nonnumeric>
[ - ]
```

where "%" means treat the number as an octal value regardless of the specified <base>.

<signed-result> output

INT:ref:1

returns the result of the conversion.

<base> input

INT:value

specifies the number base of <ascii-num>. Legitimate values are 2 through 10.

→

<status>	output
INT:ref:1	returns a number that indicates the outcome of the conversion. The values for <status> are: 1 = nonexistent number (string does not start with "+," "-," "%," or numeric) 0 = valid conversion -1 = illegal integer (number cannot be represented in 15 bits) or illegal syntax.

Condition Code Settings

The condition code has no meaning following a call to NUMIN.

Considerations

- When Number Conversion Stops

  Number conversion stops on the first ASCII numeric character representing a value greater than <base> -1 or nonnumeric ASCII character.

- Base-10 Numeric Value Range

  Base-10 numeric values must be in the range of {-32768:32767}. Numeric values in other number bases are accepted if they can be represented in 16 bits. Note that the magnitude is computed first, then the value can possibly be negated (for example, %177777 = -%1).

Example

```
CALL NUMIN ( NUMBER , RESULT , BASE , STATUS );
```

Related Programming Manual

For programming information about the NUMIN utility procedure, refer to the GUARDIAN Operating System Programmer's Guide.

NUMOUT PROCEDURE

The NUMOUT procedure converts unsigned integer values to their ASCII equivalents. The result is returned right-justified in an array. Any preceeding blanks are zero filled.

The syntax for NUMOUT is:

```
CALL NUMOUT ( <ascii-result>                    ! o
            ,<unsigned-integer>                 ! i
            ,<base>                             ! i
            ,<width> );                         ! i
```

<ascii-result> output

STRING:ref:*

is an array where the converted value returns. The ASCII representation is returned right-justified in <ascii-result>[<width> - 1]. Any preceeding blanks are zero filled.

<unsigned-integer> input

INT:value

is the value to be converted.

<base> input

INT:value

is the number base for the resulting conversion. Any number in the range 2 to 10 is valid.

<width> input

INT:value

is the maximum number of characters permitted in <ascii-result>. Characters might be truncated on the left side.

Condition Code Settings

The condition code has no meaning following a call to NUMOUT.

Considerations

- If <width> is too small to contain the number, the most significant digits are lost.

Example

```
CALL NUMOUT ( ARRAY , VARIABLE , BASE , WIDTH );
```

Related Programming Manual

For programming information about the NUMOUT utility procedure, refer to the GUARDIAN Operating System Programmer's Guide.

OPEN PROCEDURE

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

The syntax for OPEN is:

```
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
```

<filename> input

INT:ref:12

is an array containing the name (must be in internal format) of the file to be opened. Refer to the Guardian Operating System Programmer's Guide for additional information about file names.

<filenum> output

INT:ref:1

returns a number used to identify the file in subsequent system calls. A -1 is returned if OPEN fails.

<flags> input

INT:value

specifies certain attributes of the file. If omitted, all fields are set to 0. The bit fields in the <flags> parameter are defined in Table 2-3.

→

<sync-or-receive-depth> input

INT:value

The purpose of this parameter depends on the type of device being opened:

Disc file	specifies the number of nonretryable (that is, write) requests whose completion the file system must remember. A value of one or greater must be specified to recover from a path failure occurring during a WRITE operation. This value also implies the number of WRITE operations the primary process in a primary and backup process pair can perform to this file without intervening checkpoints to its backup process. If omitted, or if 0 is specified, internal checkpointing does not occur. Disc path failures are not automatically retried by the file system.
$RECEIVE file	specifies the maximum number of incoming messages read by READUPDATE that the application process is allowed to queue before corresponding REPLYs must be performed. If omitted, READUPDATE and REPLY to $RECEIVE are not permitted.
WRITE to a process pair	indicates whether or not a WRITE is automatically redirected to the backup process if the primary process or its processor module fails. If this parameter >= 1, then a WRITE is automatically redirected in a manner invisible to the originator of a message. If this parameter = 0, a WRITE cannot occur to the primary process of a process pair; an error indication returns to the message's originator. On a subsequent retry, the file system redirects the WRITE to the backup process.

For other device types, this parameter is ignored.

→

NOTE

The next two parameters are supplied only if the OPEN is by the backup process of a process pair, the file is currently open by the primary process, and the checkpointing facility is not used. Both parameters must be supplied.

<primary-filenum> input

INT:value

is the file number returned to the primary process when it opened this file. <primary-filenum> must be passed as -<filenum>.

A negative file number indicates that the same file number should be returned in the backup as was returned in the primary. If a negative file number is specified and the file number is already open by the backup process, OPEN returns file system error 12. In this situation, a NonStop process pair would indicate externally that error 12 ("file in use") exists when, in fact, the file is not in use by the normal definition (open by another process in exclusive mode).

<primary-process-id> input

INT:ref:4

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

NOTE

The next two parameters are included if the block buffer for the file is to reside in the process file segment ; otherwise, they are omitted.

If sequential block buffering is used, the file should usually be opened with protected or exclusive access. Shared access can be used, although it can cause some problems. Refer to the "Sequential Buffer Option" in the ENSCRIBE Programming Manual for your system.

→

<buffer-length> input

INT:value

is the length (in bytes) of the <sequential-block-buffer>. If the <buffer-length> is less than the <data-block-length> specified in the creation of this file or any associated alternate-key file(s), or if the file is opened with shared access, OPEN succeeds but returns a CCG indication (a subsequent call to FILEINFO returns <error> =5). The normal system buffering is then used instead of the application process's sequential buffer.

If this parameter is omitted or specified as 0, sequential buffering is not attempted.

Table 2-3. OPEN <flags> Parameter

Flag	Flag in Octal	Meaning
.<0>	%100000	must be 0 (unused). This is referred to as the high-order bit, being the leftmost.
.<1>	%40000	For the $RECEIVE file only, specifies whether or not the opener wishes to receive OPEN, CLOSE, CONTROL, SETMODE, RESETSYNC, and CONTROLBUF messages. 0 = no, 1 = yes (must be 0 for all other files).
.<2>	%20000	specifies unstructures access to an ENSCRIBE file structure, regardless of the actual file structure. Setting this bit to 0 provides normal structured access to the file. 0 = no, 1 = yes (must be 0 for all other files).
.<3>	%10000	(reserved) must be 0 for non-privileged users.
.<4:5>	%6000 (If both bits set)	access mode 0 = READ/WRITE 1 = READ-only 2 = WRITE-only 3 = reserved.
.<6>	%1000	must be 0 (unused).
.<7>	%400	must be 0 (unused).
.<8>	%200	for process files, indicates that the OPEN message is sent nowait and must be completed with a call to AWAITIO. 0 = no, 1 = yes (must be 0 for all other files).
.<9>	%100	must be 0 (unused).
.<10:11>	%60 (If both bits set)	exclusion mode 0 = shared 1 = exclusive 3 = protected.
.<12:15>	%17 (If all three bits set)	>0 implies nowait I/O and the maximum number of concurrent nowait I/O. operations that can be in progress on this file at any given time. 0 implies wait I/O.

Condition Code Settings

< (CCL) indicates that the OPEN failed (call FILEINFO). If OPEN fails a -1 will be returned in <filenum>.

= (CCE) indicates that the file opened successfully.

> (CCG) indicates the file opened successfully but an exceptional condition was detected (call FILEINFO).

Considerations

- File Numbers

  Within a process, the file numbers are unique. The lowest numeric file number is 0 and is reserved for $RECEIVE. Remaining file numbers start at 1. The lowest available file number is always assigned. Once a file is closed, its file number becomes available, and a subsequent file OPEN can reuse that file number.

- Maximum Number of Open Files

  The maximum number of files in the system that can be open at any given time depends on the space available for control blocks; access control blocks (ACBs), file control blocks (FCBs), and open control blocks (OCBs). The amount of space available for control blocks is limited primarily by the physical memory size of the system. Each process can have up to 64K bytes of space for ACBs.

- Multiple Openings by the Same Process

  If a given file is opened more than once by the same process, a new ACB is created for each OPEN. This provides logically separate accesses to the same file because a unique file number returns to the process for each OPEN. Whenever you reference a file in a procedure, the file number is supplied by you in the <filenum> parameter of the procedure.

Multiple OPENs on a given file can create a deadlock. The following shows how a deadlock situation occurs:

```
OPEN( $TERM , <filenuma> ... );
! first OPEN on process $TERM.
     .
     .
OPEN( $TERM , <filenumb> ... );
! second OPEN on process $TERM.
     .
     .
OPEN( $TERM , <filenumc> ... );
! third OPEN on process $TERM.
     .
----
d        .
e  LOCKFILE ( <filenumb>, ... );   ! the file is locked using the
a        .                         ! file number associated with
d        .                         ! the second OPEN.
l  READUPDATE ( <filenumc>, ... ); ! update the file associated
o        .                         ! with the third OPEN.
c        .
k
----
```

Locks are granted on an open file (that is, file number) basis. Therefore, if a process has multiple OPENs of the same file, a lock of one file number excludes access to the file through other file numbers. The process is suspended forever if the default locking mode is in effect.

You now have a deadlock. The file number referenced in the LOCKFILE call differs from the file number in the READUPDATE call.

- Limit of Times File Can Be Open

There is a limit to the total number of times a given file can be open at one time. This determination includes OPENs by all processes. The specific limit for a file is dependent on the file's device type:

```
Disc Files = cannot exceed 4095 OPENs per disc
Process    = defined by process
Operator   = cannot exceed 4095 OPENs
$OSP       = 127 concurrent OPENs permitted
$RECEIVE   = one OPEN per process permitted
Other      = 127 concurrent OPENs permitted.
```

- File OPENs--Errors

  If a process file is opened in a nowait manner (<flags>.<8> = 1), that file is opened as nowait and checkopened in a nowait manner. Errors detected in parameter specification and system data space allocation are returned by the call to OPEN, and the operation is considered unsuccessful. If there is an error, no message to the process being opened is sent, and no call to AWAITIO is needed to complete the OPEN.

  If there are no parameter or data space allocation errors, the <filenum> parameter is valid when OPEN returns. However, no I/O operation on the file can be initiated until the OPEN is completed, and other errors are reported by a call to AWAITIO.

  If the <tag> parameter is specified in the call to AWAITIO, a -30D returns. The values returned in the buffer and count parameters to AWAITIO are undefined. If an error returns from AWAITIO, it is the user's responsibility to close the file.

  For a nonprocess or waited (nowait depth = 0) file, <flags>.<8> is internally reset to 0 and ignored. A call to FILEINFO after the call to OPEN can return the value of the internal flags; if bit <8> = 1, then a call to AWAITIO must be performed to complete the open.

- Refer to the ENSCRIBE Programming Manual and the GUARDIAN Operating System Programmer's Guide for considerations when using nowait I/O.

- Partitioned Files

  A separate pair of FCBs exist for each partition of a partitioned file. There is one ACB per accessor (as for single-volume files), but this ACB requires more main memory since it contains the information necessary to access all of the partitions, including the location, alternate keys, and partial-key value for each partition.

- Disc File OPEN--Security Check

  When a disc file OPEN is attempted, a file security check takes place. The accessor's (that is, calling process) security level is checked against the file's security level for the requested access mode. (File security is set by the SETMODE procedure or the File Utility Program SECURE command.) If the calling process's security level is equal to or higher than the file's security level for the requested access mode, then the calling process passes the security check. If the calling process fails the security check, the OPEN fails, and a subsequent call to FILEINFO returns error 48. File security checking is shown in Figure 2-3.

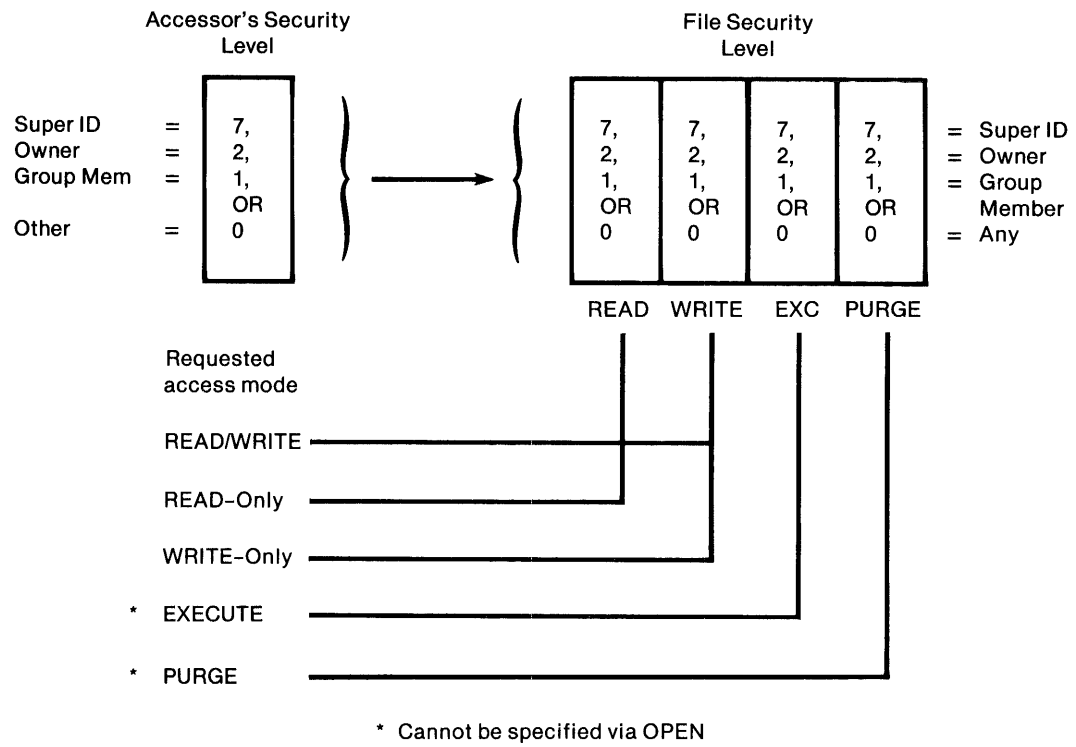


Figure 2-3. File Security Checking

- File OPEN--Exclusion and Access Mode Checking

  When a file OPEN is attempted, the requested access and exclusion modes are compared with those of any OPENs already granted for the file. If the attempted OPEN is in conflict with other OPENs, then the OPEN fails. A subsequent call to FILEINFO returns error 12. Table 2-4 lists all possible current modes and requested modes, indicating whether an OPEN succeeds or fails.

  NOTE

  Protected exclusion mode has meaning only for disc files. For other files, specifying protected exclusion mode is equivalent to specifying shared exclusion mode.

Table 2-4. Exclusion and Access Mode Checking

OPEN Attempted With		FILE CLOSED	File Currently OPEN With								
Exclusion Mode			S	S	S	E	E	E	P	P	P
	Access Mode		R / W	R	W	R / W	R	W	R / W	R	W
S	R/W	Y	Y	Y	Y						
S	R	Y	Y	Y	Y				Y	Y	Y
S	W	Y	Y	Y	Y						
E	R/W	Y	Always Fails								
E	R	Y									
E	W	Y									
P	R/W	Y		Y							
P	R	Y		Y						Y	
P	W	Y		Y							

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

Access Mode:
- R/W = READ/WRITE
- R = READ only
- W = WRITE only

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

NOTE

When a program file is running it is opened with the equivalent to R, P.

Disc File Considerations

- Maximum Number of Concurrent Nowait Operations

  The maximum number of concurrent nowait operations permitted for an OPEN of a disc file is one. Attempting to open a disc file and specify a value greater than 1 returns an error indication. A subsequent call to FILEINFO returns error 28.

- Disc Accesses and the REFRESH Option

  When a disc file that has the REFRESH option set (at CREATE time) is opened, file labels are refreshed automatically when the end-of-file (EOF) pointer is advanced. (For a description of the REFRESH option, see the section on unstructured disc files in the ENSCRIBE Programming Manual.) Depending on the particular application, there can be a significant decrease in processing throughput due to the increased number of disc WRITEs if REFRESH is set.

- Unstructured Files

  --File pointers after OPEN

  After a disc file is opened, the current-record and next-record pointers begin at a relative byte address (RBA) of zero, and the first data transfer (unless an intervening POSITION is performed) is from that location. After a successful OPEN, the pointers are:

```
current-record pointer := 0D;
next-record pointer := 0D;
```

  --Sharing the same EOF pointer

  If a given disc file is opened more than once by the same process, separate current-record and next-record pointers are provided for each OPEN, but all OPENs share the same EOF pointer.

- Structured Files

--Accessing structured files as unstructured files

The unstructured access option (<flags>.<2>) permits a file to be accessed as an unstructured file. For OPEN, with this option specified, a data transfer occurs to the position in the file specified by an RBA (instead of to the position indicated by a key address field or record number); the number of bytes transferred is that specified in the file system procedure call (instead of the number of bytes indicated by the record format). If a partitioned, structured file is opened as an unstructured file, only the first partition is opened. The remaining partitions must be opened individually with separate calls to OPEN (each OPEN specifying unstructured access).

Accessing audited structured files as unstructured files is not allowed.

CAUTION

Programmers using this option are cautioned that the block format used by ENSCRIBE must be maintained if the file is to be accessed again in its structured form. (Tandem reserves the right to change this block format at any time.) Refer to the ENSCRIBE Programming Manual for information about ENSCRIBE block formats.

- Current-State Indicators After OPEN

After successful completion of OPEN, the current-state indicators have these values:

--The current position is that of the first record in the file by primary key.

--The positioning mode is approximate.

--The comparison length is 0.

If READ is called immediately after OPEN for any structured file, it reads the first record in the file; in a key-sequenced file, this is the first record by primary key. Subsequent READs, without intervening positioning, read the file sequentially (in a relative or entry-sequenced file) or by primary key (in a key-sequenced file) through the last record in the file.

When a key-sequenced file is opened, KEYPOSITION usually is called before any subsequent I/O call (such as READ, READUPDATE, WRITE) to establish a position in the file.

Terminal Consideration

- Opening the Terminal Used as the Operator Console

  The terminal being used as the operator console should not be opened with exclusive access. If it is, console messages are not logged.

Interprocess Communication Considerations

- Maximum Concurrent Nowait Operations for an Open of $RECEIVE

  The maximum number of concurrent nowait operations permitted for an open of $RECEIVE is one. Attempting to open $RECEIVE and to specify a value greater than 1 returns an error indication. A subsequent call to FILEINFO returns error 28.

- When an OPEN Completes

  For an OPEN of another process (requesting system messages) the call to OPEN completes when the process specified to open performs a subsequent READ of its $RECEIVE file. Otherwise, the OPEN completes immediately.

Message

- Process OPEN Message

  This system message (-30) is received by a process when it is opened by another process. The process ID of the opener can be obtained in a subsequent call to LASTRECEIVE or RECEIVEINFO. Refer to Appendix F for a description and the form of this message and all system messages.

  NOTE

  This message is also received if the backup process of a process pair performs the OPEN. Therefore, a process can expect two of these messages when being opened by a process pair.

Example

```
CALL OPEN ( FILE^NAME , FILE^NUM );
```

The file in this call has the following defaults; wait I/O, exclusion mode (shared), access mode (READ/WRITE), sync depth (0).

Related Programming Manuals

For programming information about the OPEN file system procedure, refer to the GUARDIAN Operating System Programmer's Guide and the ENSCRIBE Programming Manual.

POSITION PROCEDURE

The POSITION procedure positions by primary key within relative and entry-sequenced files. For unstructured files, the POSITION procedure specifies a new current position.

For relative and unstructured files, POSITION sets the current position, access path, and positioning mode for the specified file. The current position, access path, and positioning mode define a subset of the file for subsequent access.

The POSITION procedure is not used with key-sequenced files; KEYPOSITION is used instead.

The caller is not suspended because of a call to POSITION.

A call to the POSITION procedure is rejected with an error indication if there are incomplete nowait operations pending on the specified file.

The syntax for POSITION is:

```
CALL POSITION ( <filenum>                          ! i
              ,<record-specifier> );               ! i
```

<filenum> input

INT:value

is the number of an open file that identifies the file where the positioning is to take place.

<record-specifier> input

INT(32):value

is a relative bye address (RBA) that specifies the new setting for the current-record and next-record pointers.

Relative Files: <record-specifier> is a four-byte <record-num>.

-2D specifies that the next WRITE should occur at an unused record position.

→

-1D specifies that subsequent WRITEs should be appended to the end-of-file location.

Unstructured Files: <record-specifier> is a four-byte <relative-byte-addr>.

-1D or -2D specifies that subsequent WRITEs should be appended to the EOF location.

(For relative and unstructured files, the -1D and -2D remain in effect until a new <record-specifier> is supplied.)

Entry-Sequenced Files: <record-specifier> is a four-byte <record-addr> (the primary key). Refer to the ENSCRIBE Programming Manual for information about <record-addr>.

Condition Code Settings

< (CCL) indicates that an error occurred (call FILEINFO).

= (CCE) indicates that the POSITION was successful.

> (CCG) indicates no operation; <filenum> does not designate a disc file.

Considerations

- POSITION does not cause the disc heads to be repositioned (at least until a subsequent data transfer is initiated).
- Unstructured Files

  --File pointers after POSITION

  After a successful call to POSITION for an unstructured file, the file pointers are:

```
current-record pointer := next-record pointer;
   IF rba = -1D THEN end-of-file pointer
    ELSE RBA;
```

--Value of <record-specifier> for unstructured files

Unless the unstructured file is created with the ODDUNSTR parameter set, the RBA passed in <record-specifier> must be an even number. If the ODDUNSTR parameter is set when the file is created, the RBA passed in <record-specifier> can be either an odd or an even value. (The ODDUNSTR parameter is set when the file is created, either with <filetype>.<12> of the CREATE procedure or with the Peripheral Utilities Program SET and CREATE commands.)

For even unstructured files, the <record-specifier> must be an even byte address (see CREATE procedure), or the operation fails with file system error 2.

- Relative and Entry-Sequenced Files

--Writing to entry-sequenced files

Inserts to entry-sequenced files always occur at the end of the file.

--Maximum size of relative file

POSITION cannot be used on a relative file with more than 8,388,608 blocks (this is the approximate number that can be passed and stored in the 32-bit integer <record-specifier> parameter). Attempting to do so results in file system error 23 (illegal disc address).

--Current-state indicators for structured files

After a successful POSITION to a relative or entry-sequenced file, the current-state indicators are:

Current position is that of the record indicated by the <record-specifier>.

Positioning mode is approximate.

Comparison length is 4.

Current primary-key value is set to the value of the <record-specifier>.

Example

```
CALL POSITION ( FILE^NUM , 4096D );
```

Related Programming Manuals

For programming information about the POSITION file system procedure, refer to the ENSCRIBE Programming Manual.

POSITION^SCREEN PROCEDURE

POSITION^SCREEN places control characters into the application program's I/O buffer so that the cursor positions over the first character of a data-entry field.

The syntax of POSITION^SCREEN is:

```
<num-chars> := POSITION^SCREEN ( @<screen-name>          ! i
                               , SCREEN                  ! o
                               , <buffer>                ! o
                               , <field-name> );         ! i
```

<num-chars> returned value

INT

returns the number of control characters placed into the application program's I/O buffer.

<screen-name> input

INT:value

is the address of the READ-only array that has the form definition (refer to the ENTRY Screen Formatter Operating and Programming Manual or the ENTRY520 Screen Formatter Operating and Programming Manual for an explanation of "form definition").

SCREEN output

STRING:ref:*

is the required array named SCREEN where the entry data is placed. You can access all fields using the Tandem Application Language (TAL) SCAN statement because all fields are null-terminated. Each field has no leading and trailing blanks: in other words, each field is left-justified.

→

<buffer>	output
STRING:ref:*	
is the program's I/O buffer where the control sequence is placed. The sequence is six characters long.	
<field-name>	input
STRING:ref:*	
is the name of the field where you want the cursor positioned.	

Condition Code Settings

The condition code has no meaning following a call to POSITION^SCREEN.

Example

```
NUM^CHARS := POSITION^SCREEN ( @X , SCREEN , BUF , X^NAME);
```

Related Programming Manuals

For programming information about the POSITION^SCREEN entry procedure, refer to the ENTRY Screen Formatter Operating and Programming Manual and the ENTRY520 Screen Formatter Operating and Programming Manual.

PRINTCOMPLETE PROCEDURE

The PRINTCOMPLETE procedure is used by a print process to communicate with the spooler supervisor.

The PRINTCOMPLETE procedure obtains a message from the supervisor.

The syntax for PRINTCOMPLETE is:

```
<error-code> := PRINTCOMPLETE ( <filenum-to-supervisor>        ! i
                              ,<print-control-buffer>  );    ! o
```

<error-code> returned value

INT

returns the following spooler error code:

%3000
- %3377 = supervisor file error (<8:15> contains a file error). This error indicates a communication problem with the supervisor. A print process receiving this error can call ABEND, retry the operation a number of times, or continue reading and printing jobs without any further communication with the supervisor.

%14015 = process is not a spooler supervisor.

<filenum-to-supervisor> input

INT:value

is the number of an open supervisor file (which is obtained in a previous call to OPEN).

<print-control-buffer> output

INT:ref:64

on return, contains a message from the supervisor.

Condition Code Settings

The condition code has no meaning following a call to PRINTCOMPLETE (see the <error-code> parameter).

Considerations

- PRINTCOMPLETE should not be used by a perusal process. The print process operates inconjuction with, and under the control of, the supervisor, while a perusal process operates on its own.
- The message returned by PRINTCOMPLETE is interpreted through a call to PRINTREADCOMMAND.
- PRINTCOMPLETE must be called immediately following a completion of the call on the file to the supervisor.
- Calling AWAITIO

  In addition to obtaining the supervisor message, PRINTCOMPLETE also initiates a nowait operation to the supervisor file. Thus, a call to AWAITIO must be issued to the supervisor file at some time subsequent to a call to PRINTCOMPLETE.

Example

```
PRINT^ERROR := PRINTCOMPLETE ( FILENUM^SUP , PRINT^BUFF );
```

Related Programming Manuals

For more information about the PRINTCOMPLETE Spooler procedure, refer to the Spooler Programmer's Guide.

PRINTINFO PROCEDURE

The PRINTINFO procedure is used in print processes to communicate with the spooler supervisor.

The PRINTINFO procedure returns information to the supervisor regarding a job in response to a "send status" request.

The syntax for PRINTINFO is:

```
<error-code> := PRINTINFO ( <job-buffer>                    ! i
                          ,[ <copies-remaining> ]           ! o
                          ,[ <current-page> ]               ! o
                          ,[ <current-line> ]               ! o
                          ,[ <lines-printed> ] );           ! o
```

<error-code> returned value

INT

returns the following spooler error code:

%10001 = parameter present, but its content is wrong.

<job-buffer> input

INT:ref:560

contains control information for the job started. PRINTINFO interprets the contents of <job-buffer>.

<copies-remaining> output

INT:ref:1

returns the number of copies of the job that are left to print, including the current copy.

<current-page> output

INT:ref:1

returns the number of the current page.

→

<current-line>	output
INT:ref:1	
returns the current line of the current page being printed.	
<lines-printed>	output
INT:ref:1	
returns the total number of lines printed for this copy of the job.	

Condition Code Settings

The condition code has no meaning following a call to PRINTINFO (see the <error-code> parameter).

Considerations

- PRINTINFO should not be used by a perusal process. A print process operates inconjuction with, and under the control of, the supervisor, while a perusal process operates on its own.
- PRINTINFO is used by a print process (PRINTSTART or PRINTREAD procedure) to respond to a status request from the supervisor.
- <line-printed> Parameter

  The <lines-printed> parameter is not always an indication of how many lines remain to be printed on a job because it includes lines that are printed more than once as a result of a page skip action.

Example

```
INFO^ERROR := PRINTINFO ( JOB^BUFF , COPIES );
```

Related Programming Manual

For more information about the PRINTINFO spooler procedure, refer to the Spooler System Management Guide.

PRINTINIT PROCEDURE

The PRINTINIT procedure is used in print processes to initialize communication with the spooler supervisor.

The PRINTINIT procedure initializes the print process's print control buffer that is used in calls to other print procedures.

The syntax for PRINTINIT is:

```
<error-code> := PRINTINIT ( <filenum-to-supervisor>          ! i
                          ,<print-control-buffer> );         ! i, o
```

<error-code> returned value

INT:value

returns one of the following spooler error codes:

%2000
- %2377 = file error on data file (bits <8:15> contain a GUARDIAN file system error number).

%3000
- %3377 = supervisor file error (<8:15> contains a file error). This error indicates a communication problem with the supervisor. A print process receiving this error can call ABEND, retry the operation a number of times, or continue reading and printing jobs without any further communication with the supervisor.

%4000
- %4377 = device error sent to the supervisor by the print process (bits <8:15> contain a file system error number).

%10000 = missing parameter
%10001 = parameter present, but its content is wrong.

<filenum-to-Supervisor> input

INT:value:1

is the number of an open supervisor file (that is obtained by a previous call to OPEN). This file must be opened nowait.

→

<print-control-buffer> input, output INT:ref:64 is formatted by PRINTINIT and should be passed unaltered to other print procedures.

Condition Code Settings

The condition code has no meaning following a call to PRINTINIT (see the <error-code> parameter).

Considerations

- PRINTINIT should not be used by a perusal process. A print process operates inconjuction with, and under the control of, the supervisor, while a perusal process operates on its own.
- Before calling PRINTINIT, a print process must have a file open to the supervisor with nowait I/O and a sync depth of, at most, one.
- PRINTINIT must be followed at some point with a call to AWAITIO.
- <print-control-buffer> and the Supervisor

  The <print-control-buffer> returned by PRINTINIT is used by the supervisor to send messages to the print process. The print process should never alter this buffer except with calls to the print procedures.
- Usually, PRINTINIT is called only once by a print process.

Example

```
INIT^ERROR := PRINTINIT ( FILENUM^SUP , PRINT^BUFFER );
```

Related Programming Manual

For more information about the PRINTINIT spooler procedure, refer to the Spooler Programmer's Guide.

PRINTREAD PROCEDURE

The PRINTREAD procedure can be used in print and perusal processes to access spooled data and to allow print processes to communicate with the spooler supervisor.

The PRINTREAD procedure returns one line of spooled data.

The syntax for PRINTREAD is:

```
<error-code> := PRINTREAD ( <job-buffer>              ! i, o
                          ,<data-line>                ! o
                          ,<read-count>               ! i
                          ,[ <count-read> ]           ! o
                          ,[ <pagenum> ]   ) ;        ! i
```

<error-code> returned value

INT

returns a spooler error code. Certain nonzero <error-code>s from PRINTREAD have special significance:

- %12000 = end of file. All lines in the job have been transferred (send an "end job" message to the supervisor by PRINTSTATUS; only for print processes not for perusal processes)
- %12001 = end of copy
- %12002 = invalid data file
- %12003 = CONTROL found
- %12004 = SETMODE found
- %12005 = CONTROLBUF found

Refer to the System Messages Manual for a list of spooler errors and their meanings.

<job-buffer> input, output

INT:ref:560

is the job buffer for the job being read.

→

<data-line> output

INT:ref:450 (or less)

returns a line of spooled data.

<read-count> input

INT:value

specifies the maximum number of bytes to be read.

<count-read> output

INT:ref:1

is the number of bytes actually read.

<pagenum> input

INT:value

returns one of the following:

>	0	PRINTREAD returns the first line on the page specified by this parameter.
<	0	PRINTREAD repeats the last line returned.
=	0	or absent, PRINTREAD returns the next sequential line.

Condition Code Settings

The condition code has no meaning following a call to PRINTREAD (see the <error-code> parameter).

Considerations

- Size of <data-line>

  The size of <data-line> never exceeds 450 words; however, in most cases, it is smaller.

- Critical Errors Returned From PRINTREAD

  Errors returned from PRINTREAD other than %12000-%12001 and %12003-%12005 are critical. In the case of a print process (not for perusal processes), the error should be sent to the supervisor using PRINTSTATUS.

Example

```
READ^ERROR := PRINTREAD ( JOB^BUFF , LINE , COUNT , COUNT^READ );
```

Related Programming Manuals

For programming information about the PRINTREAD spooler procedure, refer to the Spooler Programmer's Guide.

PRINTREADCOMMAND PROCEDURE

The PRINTREADCOMMAND procedure can be used in print and perusal processes to access spooled data and to allow print processes to communicate with the spooler supervisor.

The PRINTREADCOMMAND procedure interprets the information contained in the print control buffer returned from a call to PRINTCOMPLETE (in a print process) or SPOOLEREQUEST (in a perusal processs).

The syntax for PRINTREADCOMMAND is:

```
<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> returned value

INT

returns one of the following spooler error codes:

%10000 = missing parameter
%10001 = parameter is present, but its content is wrong.

<print-control-buffer> input

INT:ref:64

is passed exactly as it is returned from PRINTCOMPLETE or SPOOLEREQUEST.

→

<controlnum> output

INT:value

specifies the action requested by the supervisor:

0 = open device
1 = close device
2 = start job on device
3 = stop job on device
4 = resume job on device
5 = suspend job on device
6 = print form alignment template on device
7 = skip to page
8 = skip over pages
9 = send status of job printing on device

<device> output

INT:ref:12

specifies the particular device referenced by the control number.

<devflags> output

INT:ref:1

indicates the state of the device's truncation and header flags:

flags.<9> = batch header: 1 = on, 0 = off
flags.<10> = truncation flag: 1 = on, 0 = off
flags.<13> = header flag: 1 = on, 0 = off

All other bits are reserved for use by the spooler.

<devparam> output

INT:ref:1

is the parameter specified in the "DEV, PARM" SPOOLCOM command.

→

<devwidth> output

INT:ref:1

is the device width specified in the "DEV, WIDTH" SPOOLCOM command.

<skipnum> output

INT:ref:1

returns the number of pages to skip. The meaning of this number depends on the control number:

<controlnum> = 7 (skip to page). <skipnum> specifies the page that should be skipped to.

<controlnum> = 8 (skip over pages). <skipnum> specifies the number of pages relative to the current page that should be skipped.

<controlnum> is neither 7 nor 8. The <skipnum> parameter has no meaning.

<data-file> output

INT:ref:12

is the data file in which the job is stored if the control number is 2 (start job); otherwise, this parameter has no meaning.

<jobnum> output

INT:ref:1

is the number of the job referenced if the control number is 2 (start job); otherwise, this parameter has no meaning.

<location> output

INT:ref:8

is the location of the job started if the control number is 2 (start job); otherwise, this parameter has no meaning.

→

<form-name> output

INT:ref:8

is the form name of the job being referenced if the control number is 2 (start job); otherwise, this parameter has no meaning.

<report-name> output

INT:ref:8

is the report name of the job referenced if the control number is 2 (start job); otherwise, this parameter has no meaning.

<pagesize> output

INT:ref:1

is the page size of the job referenced if the control number is 2 (start job); otherwise, this parameter has no meaning.

Condition Code Settings

The condition code has no meaning following a call to PRINTREADCOMMAND (see the <error-code> parameter).

Considerations

- Calling PRINTREADCOMMAND Twice

  If desired, PRINTREADCOMMAND can be called once to get the <controlnum> and then a second time to get whatever particular information is needed.

- Printing a Header Message

  The print process can use the <location>, <form-name>, <report-name>, and <flags> to print out a header message.

- Passing <skipnum> Directly to PRINTREADCOMMAND

  If the control number is 7, the <skipnum> parameter can pass directly to PRINTREAD.

- Control Number 8 and PRINTREADCOMMAND

  If the control number is 8, PRINTINFO must be called to get the <current-page>. <skipnum> must then be added to the <current-page> to find the page number to pass to PRINTREAD.

- A print process may ignore the header and truncation flags and the <devwidth> parameter.

Example

```
READ^ERROR := PRINTREADCOMMAND ( PRINT^BUFFER    ! print buffer.
                               , CNTRL^NUM       ! control number.
                               ,                 ! device.
                               ,                 ! device flags.
                               ,                 ! device parameter.
                               ,                 ! device width
                               ,                 ! pages to skip.
                               , DATA^FILE );    ! data file.
```

Related Programming Manual

For programming information about the PRINTREADCOMMAND spooler procedure, refer to the Spooler Programmer's Guide.

PRINTSTART PROCEDURE

The PRINTSTART procedure formats the job buffer for a spooler job being started. The buffer is used in subsequent calls to PRINTREAD.

The syntax for PRINTSTART is:

```
<error-code> := PRINTSTART ( <job-buffer>                    ! o
                           ,<print-control-buffer>           ! i
                           ,<data-filenum>  );               ! i
```

<error-code> returned value

INT

returns one of the following spooler error codes:

0 = successful operation

%2000
-%2377 = file error on data file (bits <8:15> contain a GUARDIAN file system error number).

%3000
- %3377 = supervisor file error (<8:15> contains a file error). This error indicates a communication problem with the supervisor. A print process receiving this error can call ABEND, retry the operation a number of times, or continue reading and printing jobs without any further communication with the supervisor.

%4000
- %4377 = device error sent to the supervisor by the print process (bits <8:15> contain a file system error number).

%10001 = parameter present, but its content is wrong.

<job-buffer> output

INT:ref:560

contains control information for the job being started in a form suitable for passing to other print procedures.

→

<print-control-buffer> input

INT:ref:64

is the buffer obtained from PRINTCOMPLETE or SPOOLEREQUEST.

<data-filenum> input

INT:value

is the file number of the data file containing the started job.

Condition Code Settings

The condition code has no meaning following a call to PRINTSTART (see the <error-code> parameter).

Considerations

- <job-buffer> Also Stores Data

  In addition to containing control information for the job, the the PRINTREAD procedure uses <job-buffer> to store a block of spooled data.

- PRINTSTART is called once for each job started on a device.

- The job buffer should not be altered by the print or perusal process.

Example

```
START^ERROR := PRINTSTART ( JOB^BUFF , PRINT^BUFF , FILENUM );
```

Related Programming Manuals

For programming information about the PRINTSTART spooler procedure, refer to the Spooler Programmer's Guide.

PRINTSTATUS PROCEDURE

The PRINTSTATUS procedure can be used in print processes to communicate with the supervisor and to send an unsolicited status message to the spooler supervisor.

The syntax for PRINTSTATUS is:

```
<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
```

<error-code> returned value

INT:value

returns one of the following spooler error codes:

%2000
- %2377 = file error on data file (bits <8:15> contain a GUARDIAN file system error number).

%3000
- %3377 = supervisor file error (<8:15> contains a file error). This error indicates a communication problem with the supervisor. A print process receiving this error can call ABEND, retry the operation a number of times, or continue reading and printing jobs without any further communication with the supervisor.

%4000
- %4377 = device error sent to the supervisor by the print process (bits <8:15> contain a file system error number).

→

<filenum-to-supervisor> input

INT:value

is a number of an open supervisor file obtained in a previous call to OPEN.

<print-control-buffer> input

INT:ref:64

is the buffer obtained from the PRINTCOMPLETE procedure.

<msg-type> input

INT:value

specifies the type of message being sent, as follows:

0 = sending status of job
1 = error occurred on print device; previous operation was unsuccessful
2 = end of job
3 = unable to open device
4 = invalid operation in this state
5 = error occurred on print device; previous operation was successful

<device> input

INT:ref:12

is the name of the device on which an error occurs.

<error> input

INT:value

is the error that caused this call to PRINTSTATUS. It is sent to the supervisor.

%4000 = device error sent to the supervisor by the print
%4377 process (bits <8:15> contain a GUARDIAN file system error number).

→

```
    %13000 = no such device
    %13001 = device already open
    %13002 = No job on device
    %13003 = Job is running
    %13004 = "table is full."  It is sent by a print process
             to the supervisor when the print process is already
             handling as many jobs as it can, and the supervisor
             instructs it to start another job.
```

Refer to the System Messages Manual for a complete list and description of spooler errors.

<num-copies> input

INT:value

is the number of copies of the job remaining to be printed.

<page> input

INT:value:1

is the current page.

<line> input

INT:value

if present, is the current line (from PRINTINFO).

<lines-printed> input

INT:value

is the number of lines printed.

Condition Code Settings

The condition code has no meaning following a call to PRINTSTATUS (see the <error-code> parameter).

Considerations

- PRINTSTATUS should not be used by a perusal process. A print process operates in conjunction with, and under the control of, the supervisor, while a perusal process operates on its own.

- Message Type and Parameters of PRINTSTATUS

  The file number to supervisor, print control buffer, message type, and device are required parameters and must always be present in a call to PRINTSTATUS. The remaining parameters are optional: PRINTSTATUS might need these parameters, however, depending on the message type.

  Table 2-5 shows which parameters are needed for each message type:

Table 2-5. PRINTSTATUS Message Type and Parameters

	Error	Copies	Page	Lines Printed
0		X	X	X
1	X			
2				
3				
4	X			
5	X			

- PRINTSTATUS is a nowait operation and must be completed with a call to AWAITIO.

- Message types 1 and 5 inform the supervisor of an error occurring on a print device.

  Message type 5 is sent if the previous operation on the device is successful. When it receives the message, the supervisor can instruct the print process to retry the operation. If the operation fails again, the print process sends message type 1, which indicates to the supervisor that a retry of an operation failed.

  Message type 5 causes the supervisor to reset its retry count for that device.

Example

```
STATUS^ERROR := PRINTSTATUS ( FILENUM^SUP
                            , PRINT^BUFF
                            , MSG
                            , DEVICE
                            ,                ! error.
                            ,                ! number of copies.
                            , PAGE
                            ,                ! line.
                            , NUM^LINES );
```

Related Programming Manual

For programming information about the PRINTSTATUS spooler procedure, refer to the Spooler Programmer's Guide.

PRIORITY PROCEDURE

The PRIORITY procedure enables a process to examine or change its initial priority and current priority.

The syntax for PRIORITY is:

```
{ <old-priority> := } PRIORITY ( [ <new-priority> ]          ! i
{ CALL             }            ,[ <init-priority> ] );     ! o
```

<old-priority> returned value

INT

returns a value that is either:

- The current priority of the process if <new-priority> is not specified
- The previous value of the current priority when <new-priority> is specified
- A 0, indicating that the <new-priority> value specified as a value is out of range and that the priority was not changed

<priority> input

INT:value

specifies a new execution priority value in the range of {1:199} for this process. If omitted, the current priority remains unchanged (the range for a privileged caller is {1:255}).

<init-priority>

INT:ref:1 output

returns the initial run priority of the process when it was started.

Condition Code Settings

The condition code has no meaning following a call to PRIORITY.

Considerations

- Privileged Calls to PRIORITY

  A caller of PRIORITY who is executing in privileged mode can set its priority to a value greater than 199. However, if such a process has a priority greater than that of the memory manager process and gets a memory page fault, the process is aborted with a "no memory available," trap number 12.

- Why Current Rather Than Assigned Priority Returned

  The current priority rather than the initial priority is returned. Due to the sliding priority feature on the NonStop system, the current priority may be lower than the initial priority if the process is CPU-bound (that is, the process does not perform any I/O requests while running.)

Example

```
LAST^PRI := PRIORITY ( 100 );     ! changes the current priority to
                                  ! 100.
```

Related Programming Manuals

For programming information about the PRIORITY process control procedure, refer to the GUARDIAN Operating System Programmer's Guide.

PROCESSACCESSID PROCEDURE

The PROCESSACCESSID procedure is used to obtain the accessor ID of the calling process.

The syntax for PROCESSACCESSID is:

```
<accessor-id> := PROCESSACCESSID;

<accessor-id>                 returned value

  INT

  returns the accessor ID of the caller in the following form:

     <accessor-id>.<0:7>   = group number
                  .<8:15>  = user number
```

Condition Code Settings

The condition code has no meaning following a call to PROCESSACCESSID.

Considerations

- When Accessor IDs Differ

  For a given process, the accessor ID returned from the PROCESSACCESSID procedure is normally the same as that returned from the CREATORACCESSID procedure. The accessor IDs differ only when a program file is run with "set accessor ID to program file's owner ID" specified (usually with the File Utility Program (FUP) SECURE command and PROGID option). In that case, the accessor ID returned by PROCESSACCESSID is the same as that of the program file's owner. Refer to the GUARDIAN Operating System Utilities Reference Manual for information about FUP SECURE.

Example

```
MY^ACCESSOR^ID := PROCESSACCESSID;
```

Related Programming Manual

None

PROCESSFILESECURITY PROCEDURE

The PROCESSFILESECURITY procedure is used to examine or set the file security for the current process. This is the security used for any file creation attempts following a call to PROCESSFILESECURITY.

The syntax for PROCESSFILESECURITY is:

```
<old-security> := PROCESSFILESECURITY ( <security> );      ! i

<old-security>                returned value

  INT

  is the old file security.

<security>                    input

  INT:value

  is the new file security.  The security bits are:

    <security>.<0:3>   = 0
              .<4:6>   = ID allowed for READ
              .<7:9>   = ID allowed for WRITE
              .<10:12> = ID allowed for EXECUTE
              .<13:15> = ID allowed for PURGE

  ID can be one of the following:

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

  Refer to the GUARDIAN Operating System Utilities Reference Manual
  for more information about security.
```

Condition Code Settings

The condition code has no meaning following a call to PROCESSFILESECURITY.

Example

```
OLD^SECURITY := PROCESSFILESECURITY ( SECURITY );
```

Related Programming Manual

None

PROCESSINFO PROCEDURE

The PROCESSINFO procedure is used to obtain process status information.

The syntax for PROCESSINFO is:

```
{ <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> ]                   ! o
                             ,[ <processtime> ]                 ! o
                             ,[ <waitstate> ]                   ! o
                             ,[ <process-state> ]               ! o
                             ,[ <library-filename> ]            ! o
                             ,[ <swap-filename> ]  );           ! o
```

<error> returned value

INT

returns a value indicating the outcome of the call.

0 = status for process <cpu,pin> returns.

1 = process <cpu,pin> does not exist or does not match specified criteria (see <search-mode>). Status for next higher <cpu,pin> in the specified processor is returned. The process ID (PID) of the process for which status is being returned returns, in the <process-id> parameter (if present).

2 = process <cpu,pin> does not exist, and no higher <cpu,pin> in the specified processor that matches the specified criteria exists (see <search-mode>).

3 = unable to communicate with <cpu>.

4 = <cpu> does not exist.

→

5 = the system specified by <sysnum> could not be accessed.

99 = parameter error.

<cpu,pin> input

INT:value

is the processor and pin number of the process whose status is being requested.

<process-id> input, output

INT:ref:4

is an array where PROCESSINFO returns the PID of the process whose status is actually being returned. This can be different from the process whose status is requested through <cpu,pin> (see the <error> parameter).

On input, the <process-id> contents can be used as a search criterion (see the <search-mode> parameter).

<creator-accessor-id> input, output

INT:ref:1

returns the creator accessor ID of <process-id>. The creator accessor ID identifies the user who initiates the creation of the process. Refer to the GUARDIAN Operating System Programmer's Guide for information about creator accessor ID.

On input, the <creator-accessor-id> contents can be used as a search criterion (see the <search-mode> parameter).

<process-accessor-id> input, output

INT:ref:1

returns the process accessor ID of <process-id>. Refer to the GUARDIAN Operating System Programmer's Guide for additional information about process accessor ID.

On input, the <process-accessor-id> contents can be used as a search criterion (see the <search-mode> parameter).

→

<priority> input, output

INT:ref:1

returns the current execution priority of this process.

On input, the <priority> contents can be used as a search criterion (see the <search-mode> parameter).

<program-filename> input, output

INT:ref:12

is an array where PROCESSINFO returns the name of the <process-id>'s program file.

On input, the <program-filename> contents can be used as a search criterion (see the <search-mode> parameter).

<home-terminal> input, output

INT:ref:12

is an array where PROCESSINFO returns the device name of <process-id>'s home terminal.

On input, the <home-terminal> contents can be used as a search criterion (see the <search-mode> parameter).

<sysnum> input

INT:value

specifies the system (in a network) where the process for which information is to be returned is running. If this parameter is omitted, the local system is assumed.

<search-mode> input

INT:value

is a bit mask that specifies one or more "search" conditions.

→

On the call, the parameters to PROCESSINFO contain the values for the search conditions; that is, the information supplied in the output parameters is what the <search-mode> parameter uses in order to obtain a process's status. The bit fields in <search-mode> specify the conditions being searched for:

```
If <search-mode>.<0> = 1  must match <process-id> for 3 words
                    = 0  no search

                .<1> = 1  must match <creator-accessor-id>
                    = 0  no search

                .<2> = 1  must match <process-accessor-id>
                    = 0  no search

                .<3> = 1  must be <= <priority>
                    = 0  no search

                .<4> = 1  must match <program-filename>
                    = 0  no search

                .<5> = 1  must match <home-terminal>
                    = 0  no search
```

If multiple search conditions are specified, then all must be met.

If <search-mode> is omitted, a 0 is used.

<priv-only> output

INT:ref:*

This parameter can only be used by a privileged caller.

<process-time> output

FIXED:ref:1

returns the process time, in microseconds, for which the process has executed.

→

If you are using the PROCESSINFO procedure to obtain process time of a process that is running on a NonStop 1+ system, then <process-time> returns 0F.

<wait-state> output

INT:ref:1

returns the wait field indicating what, if anything, the process is waiting on. It is obtained from the wait field of the awake/wait word in the process's process control block. The following bits are defined:

```
<wait-state>.<8>  wait on PON (CPU power on)
            .<9>  wait on IOPON (I/O power on)
            .<10> wait on INTR (interrupt)
            .<11> wait on LINSP (INSPECT event)
            .<12> wait on LCAN (message system, cancel)
            .<13> wait on LDONE (message system, done)
            .<14> wait LTMF (TMF request)
            .<15> wait on LREQ (message system, request)
```

The bits in the wait field are numbered from left to right; thus, if octal 3 (%003) appears, this means that bits 14 and 15 are set.

<process-state> output

INT:ref:1

returns the state of the process specified by <cpu,pin>. The bits are defined as follows:

```
.<0>      privileged process
.<1>      page fault occured
.<2>      process is on the ready list
.<3>      system process
.<4:5>    reserved
.<6>      MAB in system code
.<7>      process not accepting any messages
.<8>      temporary system process
.<9>      process has logged on (called VERIFYUSER)
.<10>     in a pending process state
```

→

.<11:15> the process state, where:

```
0 = unallocated
1 = starting
2 = runnable
3 = suspended
4 = DEBUG mab
5 = DEBUG breakpoint
6 = DEBUG trap
7 = DEBUG request
8 = INSPECT mab
9 = INSPECT breakpoint
10 = INSPECT trap
11 = INSPECT request
12 = SAVEABEND
13 = terminating
```

If you are using PROCESSINFO to obtain process status on a process that is running on a NonStop 1+ system, then <process-state> returns 0.

<library-filename> output

INT:ref:12

returns the name of the library file used by the process. If the process does not have an associated library file, then <library-filename> is blank-filled.

If you are using PROCESSINFO to obtain process status on a process that is running on a NonStop 1+ system, then <library-filename> returns blanks.

<swap-filename> output

INT:ref:12

returns the name of the swap file for the process's code segment. Normally, this is the name of a temporary file unless a specific swap file is supplied at run time.

If you are using PROCESSINFO to obtain process status on a process that is running on a NonStop 1+ system, then <swap-filename> returns blanks.

Condition Code Settings

The condition code has no meaning following a call to PROCESSINFO.

Considerations

- Remote or Local Form of <process-id>

  If <sysnum> specifies a remote system, <process-id> returns in network form; otherwise, <process-id> returns in local form.

- Remote System <sysnum>--File Name Local Form

  If <sysnum> specifies a remote system, file names (such as home terminal) return in local form (starting with "$").

- CPU PID Field

  Although there are 8 bits in the CPUPID field, the call to PROCESSINFO only looks at bits <3:7>. You should not use bits <0:2>, because the operating system assumes that these bits are unused (that is, 0).

Example

```
CALL PROCESSINFO ( PIN , PID , CAID , PAID , PRI , PROG
                   , HOMETERM , , MODE );
```

Related Programming Manual

None

PROCESSORSTATUS PROCEDURE

The PROCESSORSTATUS procedure is used to obtain a count of the number of processor modules in a system and their operational states.

The syntax for PROCESSORSTATUS is:

```
<processor-status> := PROCESSORSTATUS;


<processor-status>          returned value

  INT(32)

  returns two words indicating the count and state of processor
  modules (CPUs).

  The most significant word is the maximum configured count of
  processor modules plus one.

  The least significant word is a bit mask indicating the
  operational state of each processor module:

                 ----------------------------------------
  Word  [0]     | most significant word, count of CPUs   |
                |----------------------------------------|
        [1]     | least significant word, bit mask 1 or 0 |
                 ----------------------------------------

        <ls word>.<0>  = processor module 0
        <ls word>.<1>  = processor module 1
             .
             .
             .
        <ls word>.<14> = processor module 14
        <ls word>.<15> = processor module 15

  For each bit:

  1 = up    indicates that the corresponding processor module is
            up (operational).
  0 = down  indicates that the corresponding processor module is
            down or does not exist.
```

Condition Code Settings

The condition code has no meaning following a call to PROCESSORSTATUS.

Example

```
PROCESSOR^STAT := PROCESSORSTATUS;
```

Related Programming Manual

None

PROCESSORTYPE PROCEDURE

The PROCESSORTYPE procedure returns the processor type of a specified system and CPU.

The syntax for PROCESSORTYPE is:

```
<type> := PROCESSORTYPE ( [ <cpu> ]                          ! i
                         ,[ <sysid> ]  );                    ! i


<type>                        returned value

  INT

  returns one of the following values:

    -2 = feature not supported for the system named in <sysid>
    -1 = unable to communicate with CPU (either it does not exist
         or the network is down)
     0 = NonStop 1+ processor
     1 = NonStop processor
     2 = TXP processor

  If <cpu> is greater than 16 or less than 0, then -1 is
  returned.  If <sysid> is invalid or the system is unavailable
  across the network, then -1 is returned.


<cpu>                         input

  INT:value

  is the CPU number of the processor of which the type is
  returned.

  If no value is specified for <cpu>, the CPU from which the
  call is made is used and the <sysid> parameter is ignored.


<sysid>                       input

  INT:value

  is the system number, identifying the system of the processor
  of which the type is returned.  If no value is specified for
  <sysid> the system from which the call is made is used.
```

Condition Code Settings

The condition code has no meaning following a call to PROCESSORTYPE.

Example

```
TYPE^CPU := PROCESSORTYPE ( PROCESSOR , SYSTEM^NUM );
```

Related Programming Manual

None

PROCESSTIME PROCEDURE

The PROCESSTIME procedure returns the process execution time of any process in the network.

The syntax for PROCESSTIME is:

```
<process-time> := PROCESSTIME ( [ <cpu,pin> ]          ! i
                                ,[ <sysid> ] );        ! i
```

<process-time> returned value

FIXED

is the process execution time of the specified process in the network.

-1F = indicates that the process does not exist.

-2F = indicates that the system is unavailable or does not exist. The procedure cannot get resources (link control blocks) to execute, or the system is running on an earlier operating system version than B00.

>= 0F value indicates that PROCESSTIME was successful.

<cpu,pin> input

INT:value

is the processor and pin of the process whose execution time is to be returned. If <cpu,pin> is omitted, the <cpu,pin> of the current process (calling process) is used, even if <sysid> is different than the current system.

<sysid> input

INT:value

is the system number. <sysid> defaults to the current system.

Condition Code Settings

The condition code has no meaning following a call to PROCESSTIME (see the <process-time> parameter).

Example

```
IF ( PROCESS^TIME := PROCESSTIME ( PID , SYS^NUM )) >= 0F
   THEN ...   ! successful.
   ELSE ...   ! PROCESSTIME not available.
```

Related Programming Manual

None

PROGRAMFILENAME PROCEDURE

The PROGRAMFILENAME procedure is used to obtain the name of the calling process's program file.

The main use of this procedure is to allow a primary process to create its backup process without having to hard code the program file name into the source program.

The syntax for PROGRAMFILENAME is:

```
CALL PROGRAMFILENAME ( <program-file> );              ! o

<program-file>                output

  INT:ref:12

  is an array where PROGRAMFILENAME returns the name of the
  process's program file.
```

Condition Code Settings

The condition code has no meaning following a call to PROGRAMFILENAME.

Example

```
CALL PROGRAMFILENAME ( MYPROG );
```

Related Programming Manual

None

PURGE PROCEDURE

The PURGE procedure is used to delete a disc file that is not open. When PURGE is executed, the disc file name is deleted from the volume's directory, and any disc space previously allocated to that file is made available to other files.

The syntax for PURGE is:

```
CALL PURGE ( <filename> );            ! i

<filename>                    input

  INT:ref:12

  is an array containing the name of the disc file to be purged.
  To purge either a permanent or temporary disc file, <filename>
  must be of the form:

  Permanent Disc File

    <filename>[0:3]  =  $<volname><blank-fill>
                                or
                        \<sysnum><volname><blank-fill>
              [4:7]  =  <subvol-name><blank-fill>
              [8:11] =  <disc-filename><blank-fill>

  Temporary Disc File

    <filename>[0:3]  =  $<volname><blank-fill>
                                or
                        \<sysnum><volname><blank-fill>
              [4:11] =  #<temporary-filename>
```

Condition Code Settings

- < (CCL) indicates that the PURGE failed (call FILEINFO). Note, however, that in the case of a disc free-space error (such as file system errors 52, 54, 58), the file is purged, and an error returns.
- = (CCE) indicates that the file purged successfully.
- > (CCG) indicates that the device is not a disc.

Considerations

- Purge Failure

  If PURGE fails, the reason for the failure can be determined by calling FILEINFO, passing -1 as the <filenum> parameter.

- Purging a File Audited by the Transaction Monitoring Facility (TMF)

  If the file is a file audited by TMF and there are pending transaction-mode record locks or file locks, any attempt to purge that file fails with file error 12, whether or not openers of the file still exist.

  When an audited file is purged, all corresponding dump records are deleted from the TMF catalog. If TMF is not active, attempts to purge an audited file fail with file system error 82.

- Security Consideration

  File purging normally is performed in a logical fashion; the data is not necessarily overwritten or erased, but rather pointers are changed to show the data to be absent. For security reasons, you might want to set the CLEARONPURGE flag for a file, using either function 1 of the SETMODE procedure or the File Utility Program SECURE command. Either way, this option causes all data to be physically erased (overwritten with zeros) when the file is purged. Refer to the ENSCRIBE Programming Manual for a description of purging data.

Example

```
CALL PURGE ( OLD^FILE^NAME );
```

Related Programming Manual

For programming information about the PURGE file system procedure, refer to the ENSCRIBE Programming Manual and the GUARDIAN Operating System Programmer's Guide.

PUTPOOL PROCEDURE

The PUTPOOL procedure returns a block of memory to a buffer pool.

The syntax for PUTPOOL is:

```
CALL PUTPOOL ( <pool-head>                          ! i, o
             ,<pool-block> );                       ! i
```

<pool-head> input, output

INT .EXT:ref:19

is the address of the pool head of the pool from which the block of memory was obtained using GETPOOL.

<pool-block> input

INT .EXT:ref:*

is the address of the block to be returned to the pool.

Condition Code Settings

< (CCL) indicates that the data structures are invalid or that <pool-block> is not a block in the buffer pool.

= (CCE) indicates that the operation is successful.

> (CCG) does not return from PUTPOOL.

Considerations

- A Bounds Violation Trap

  GETPOOL and PUTPOOL do not check pool data structures on each call. A process that destroys data structures can get a bounds violation trap on a call to GETPOOL or PUTPOOL.

Example

```
CALL PUTPOOL ( POOL^HEAD , PBLOCK );
```

POOl^HEAD is the pool head of the pool from which the block of memory was obtained, and PBLOCK is the block to be returned to the pool.

Related Programming Manual

For programming information about the PUTPOOL memory management procedure, refer to the GUARDIAN Operating System Programmer's Guide.

READ PROCEDURE

The READ procedure is used to return data from an open file to the application process's data area.

The READ procedure performs sequential reading of a disc file. For key-sequenced, relative, and entry-sequenced files, the READ procedure reads a subset of records in the file. (A subset of records is defined by an access path, positioning mode, and comparison length.)

The syntax for READ is:

```
CALL READ ( <filenum>                          ! i
          ,<buffer>                            ! o
          ,<read-count>                        ! i
          ,[ <count-read> ]                    ! o
          ,[ <tag> ]  );                       ! i
```

<filenum> input

INT:value

is the number of an open file that identifies the file to be read.

<buffer> output

INT:ref:*

is an array in the application process where the information read from the file returns.

<read-count> input

INT:value

is the number of bytes to be read:

{0:4096} for disc files (see "Considerations")
{0:32767} for nondisc files
{0:32000} for $RECEIVE

<count-read> output

INT:ref:1

is for wait I/O only. It returns a count of the number of bytes returned from the file into <buffer>.

<tag> input

INT(32):value

is for nowait I/O only. <tag> is a value you define that uniquely identifies the operation associated with this READ.

NOTE

The system stores the <tag> value until the I/O operation completes. The system then returns the <tag> information to the program in the <tag> parameter of the call to AWAITIO, thus indicating that the operation completed.

Condition Code Settings

< (CCL) indicates that an error occurred (call FILEINFO).

= (CCE) indicates that the READ is successful.

> (CCG) for disc and nondisc devices, indicates that the end of file (EOF) is encountered (no more records in this subset); for the $RECEIVE file, a system message is received (call FILEINFO).

Considerations

- Waited READ

  If a waited READ is executed, the <count-read> parameter indicates the number of bytes actually read.

- Nowait READ

  If a nowait READ is executed, <count-read> has no meaning and can be omitted. The count of the number of bytes read is obtained through the <count-transferred> parameter of the AWAITIO procedure when the I/O operation completes.

The READ procedure must complete with a call to the AWAITIO when it is used with a file that is opened nowait.

- READ From Nondisc Device

  If the READ is from a nondisc device, the right half of the last word of an odd count transfer is unwanted or incorrect information.

- READ Call When Default Locking Mode Is in Effect

  If the default locking mode is in effect when a call to READ is made to a locked file, but the <filenum> of the locked file differs from the <filenum> in the call, the caller of READ is suspended and queued in the "locking" queue behind other processes attempting to lock or read the file or record.

  NOTE

  A deadlock condition occurs if a call to READ is made by the process that opened and locked the file, but the <filenum> of the locked file differs from the <filenum> supplied to READ or READUPDATE. Refer to the OPEN procedure for an explanation of multiple OPENs of files and file numbers.

- Read Call When Alternate Locking Mode Is in Effect

  If the alternate locking mode is in effect when READ is called, and the file or record is locked through a file number other than that supplied in the call, the call is rejected with file system error 73 ("file is locked").

- Locking Mode for Read

  The locking mode is specified by the SETMODE procedure, function 4. If you encounter error 73 ("file is locked"), you do not need to call SETMODE for every READ. SETMODE stays in effect indefinitely (for example, until another SETMODE is performed or the file is closed), and there is no additional overhead involved.

Disc File Considerations

- Bad I/O Count

  For DP1 disc files:

  If the <read-count> parameter attempts to transfer more than nine sectors of data, the call is rejected with file system error 21. No single disc transfer can span more than two extents.

For DP2 disc files:

There is no restriction on the beginning file position. The BUFFERSIZE attribute value (which is set by specifying SETMODE function 93) does not constrain the allowable <read-count> in any way, however, there is a performance penalty if the READ does not start on a BUFFERSIZE boundary and have a <read-count> of <= BUFFERSIZE. DP2 disc process executes your requested I/O in (possibly multiple) units of BUFFERSIZE blocks starting on a block boundary.

- Structured Files

--Selecting a subset of records for sequential READs

The subset of records read by a series of calls to READ is specified through the POSITION or KEYPOSITION procedures.

--Sequential reading of an approximate subset of records

If an approximate subset is being read, the first record returned is the one whose key field, as indicated by the current key specifier, contains a value equal to or greater than the current key. Subsequent reading of the subset returns successive records until the last record in the file is read (an EOF indication is then returned).

--Sequential reading of a generic subset of records

If a generic subset is being read, the first record returned is the one whose key field, as designated by the current-key specifier, contains a value equal to the current key for <comparison-length> bytes. Subsequent reading of the file returns successive records whose key matches the current key (for <comparison-length> bytes). When the current key no longer matches, an EOF indication returns.

For relative and entry-sequenced files, a generic subset of the primary key is equivalent to an exact subset.

--Sequential reading of an exact subset of records

If an exact subset is being read, the only records returned are those whose key field, as designated by the current-key specifier, contains a value of exactly the comparison length bytes (see the KEYPOSITION procedure) and is equal to the key. When the current key no longer matches, an EOF indication returns. The exact subset for a key field having a unique value is at most one record.

--Current-state indicators after READ

After a successful READ, the current-state indicators have these values:

Current position	= record just read
Positioning mode	= unchanged
Comparison length	= unchanged
Current primary-key value	= set to the value of the primary-key field in the record

- Unstructured Files

--Unstructured READs

Data transfer begins from an unstructured disc file at the position indicated by the next-record pointer.

The READ procedure reads records sequentially on the basis of a beginning relative byte address (RBA) and the length of the records read.

--Number of bytes read using ODDUNSTR

If the unstructured file is created with the ODDUNSTR (odd unstructured file) parameter set, the number of bytes read is exactly the number of bytes specified with <read-count>. If the ODDUNSTR parameter is not set when the file is created, the value of <read-count> is rounded up to an even number before the READ is executed.

The ODDUNSTR parameter is set when the file is created, either with <filetype>.<12> of the CREATE procedure or with the Peripheral Utilities Program SET and CREATE commands.

--Maximum READ count

For DP1 disc files:

The maximum READ count for unstructured files depends on the current file position. To read unstructured files with a <read-count> of up to 4096 bytes, the file must be positioned on a sector boundary. Whenever the file's current position RBA is not a multiple of 512 and the count exceeds 3584 (seven sectors), error 21 (bad I/O count) returns. To calculate the maximum READ count possible, use the following formula:

```
maxcount := 4096 - (next-record pointer \ 512)
```

For DP2 disc files:

Unstructured files are transparently blocked. The BUFFERSIZE file attribute value, if not set by the user, defaults to 4096 bytes. This BUFFERSIZE attribute determines the size of the READ count possible. The file must be positioned on a 2048-byte boundary.

--Determination of <count-read> for unstructured READs

After a successful call to READ for an unstructured file, the value returned in <count-read> is determined by:

```
<count-read> := $MIN(<read-count> &
                     eof-pointer - next-record pointer)
```

--File pointers after READ

After a successful READ to an unstructured file, the file pointers are:

CCG = 1 if the next-record pointer = EOF pointer; otherwise, CCG = 0

current-record pointer = old next-record pointer

next-record pointer = old next-record pointer + <count-read>

Example

```
CALL READ ( FILE^NUM , IN^BUFFER , 72 , NUM^XFERRED );
```

The READ permits up to 72 bytes to be read into IN^BUFFER, and the count actually read returns into NUM^XFERRED.

Related Programming Manuals

For programming information about the READ file system procedure, refer to the GUARDIAN Operating System Programmer's Guide, the ENSCRIBE Programming Manual, and the data communication manuals.

READLOCK PROCEDURE

The READLOCK procedure performs sequential locking and reading of records in a disc file, exactly like the combination of LOCKREC and READ.

The syntax for READLOCK is:

```
CALL READLOCK ( <filenum>                          ! i
              ,<buffer>                            ! o
              ,<read-count>                        ! i
              ,[ <count-read> ]                    ! i
              ,[ <tag> ]  );                       ! i
```

<filenum> input

INT:value

is the number of an open file that identifies the file to be read.

<buffer> output

INT:ref:*

is an array in the application process where the information read from the file returns.

<read-count> input

INT:value

is the number of bytes read: {0:4096}.

<count-read> output

INT:ref:1

is for wait I/O only. <count-read> returns a count of the number of bytes returned from the file into <buffer>.

→

<tag> input

INT(32):value

is for nowait I/O only. <tag> is a value you define that uniquely identifies the operation associated with this READLOCK.

NOTE

The system stores the <tag> value until the I/O operation completes. The system then returns the <tag> information to the program in the <tag> parameter of the call to AWAITIO, thus indicating that the operation completed.

Condition Code Settings

< (CCL) indicates that an error occurred (call FILEINFO).

= (CCE) indicates that the READLOCK is successful.

> (CCG) indicates end of file (EOF). There are no more records in this subset.

Considerations

- Nowait and READLOCK

  If the READLOCK procedure is used to initiate an operation with a file-opened nowait, it must complete with a corresponding call to the AWAITIO procedure.

- READLOCK for Key-Sequenced, Relative, and Entry-Sequenced Files

  For key-sequenced, relative, and entry-sequenced files, a subset of the file (defined by the current access path, positioning mode, and comparison length) is locked and read with successive calls to READLOCK.

  For key-sequenced, relative, and entry-sequenced files, the first call to READLOCK after a positioning (or OPEN) locks and then returns the first record of the subset. Subsequent calls to READLOCK without intermediate positioning locks, returns successive records in the subset. After each of the subset's records are read, the position of the record just read becomes the file's current position. An attempt to read a record following the last record in a subset returns an EOF indication.

- Locking Records in an Unstructured File

  READLOCK can be used to lock record positions, represented by a relative byte address (RBA), in an unstructured file. When sequentially reading an unstructured file with READLOCK, each call to READLOCK first locks the RBA stored in the current next-record pointer and then returns record data beginning at that pointer for <read-count> bytes. After a successful READLOCK, the current-record pointer is set to the previous next-record pointer, and the next-record pointer is set to the previous next-record pointer plus <read-count>. This process repeats for each subsequent call to READLOCK.

- See "Considerations" for the READ procedure.

Example

```
CALL READLOCK ( FILE^NUM , IN^BUFFER , 72 , NUM^READ );
```

Related Programming Manual

For programming information about the READLOCK file system procedure, refer to the ENSCRIBE Programming Manual.

READ^SCREEN PROCEDURE

For users of the ENTRY or ENTRY520 screen formatter, the READ^SCREEN is used to place the control sequence required to read the screen from the terminal into the application program's I/O buffer. Normally, READ^SCREEN is used to input data entries when you have signaled with a function key that the screen data is ready for input.

The syntax for READ^SCREEN is:

```
<num-chars> :=  READ^SCREEN ( @<screen-name>              ! i
                            , <buffer> );                 ! o
```

<num-chars> returned value

INT

returns the number of control characters entered in the application program's I/O buffer.

<screen-name> input

INT:value

is the address of the READ-only array that has the form definition (refer to the ENTRY Screen Formatter Operating and Programming Manual or the ENTRY520 Screen Formatter Operating and Programming Manual for an explanation of "form definition").

<buffer>

STRING:ref:*

is the program's I/O buffer where the control sequence is placed. The READ control sequence is 2 characters long.

Condition Code Settings

The condition code has no meaning following a call to READ^SCREEN.

Example

```
NUM^CHARS := READ^SCREEN ( @X , BUF , X^IOBUF , CNT);
```

Related Programming Manuals

For programming information about the READ^SCREEN entry procedure, refer to the ENTRY Screen Formatter Operating and Programming Manual or the ENTRY520 Screen Formatter Operating and Programming Manual.

READUPDATE PROCEDURE

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

- Disc Files

  READUPDATE is used for random processing. Data is read from the file at the position of the current-record pointer. A call to this procedure typically follows a corresponding call to POSITION or KEYPOSITION. The values of the current- and next-record pointers do not change with the call to READUPDATE.

- Interprocess Communication

  READUPDATE is used to read a message from the $RECEIVE file that is answered in a later call to REPLY. Each message read by READUPDATE must be replied to in a corresponding call to REPLY.

The syntax for READUPDATE is:

```
CALL READUPDATE ( <filenum>                          ! i
                 ,<buffer>                           ! o
                 ,<read-count>                       ! i
                 ,[ <count-read> ]                   ! o
                 ,[ <tag> ]  );                      ! i
```

<filenum> input

INT:value

is the number of an open file that identifies the file to be read.

<buffer> output

INT:ref:*

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

→

<read-count> input

INT:value

is the number of bytes to be read.

```
disc files = {0:4096}
$RECEIVE   = {0:32000}
```

<count-read> output

INT:ref:1

is for wait I/O only. <count-read> returns a count of the number of bytes returned from the file into <buffer>.

<tag> input

INT(32):value

is for nowait I/O only. <tag> is a value you define that uniquely identifies the operation associated with this READUPDATE.

NOTE

The system stores the <tag> value until the I/O operation completes. The system then returns the <tag> information to the program in the <tag> parameter of the call to AWAITIO, thus indicating that the operation completed.

Condition Code Settings

< (CCL) indicates that an error occurred (call FILEINFO).

= (CCE) indicates that the READUPDATE is successful.

> (CCG) indicates that a system message is received through $RECEIVE. (CCG does not return from READUPDATE for disc files.)

Considerations

- Random Processing and Positioning

  A call to READUPDATE returns the record from the current position in the file. Because READUPDATE is designed for random processing, it cannot be used for successive positioning through a subset of records like the READ procedure. Rather, READUPDATE is used to read a record after a call to POSITION or KEYPOSITION, possibly in anticipation of a subsequent update through a call to the WRITEUPDATE procedure.

- Calling READUPDATE After READ

  A call to READUPDATE after a call to READ, without intermediate positioning, returns the same record as the call to READ.

- Waited READUPDATE

  If a waited READUPDATE is executed, the <count-read> parameter indicates the number of bytes actually read.

- Nowait READUPDATE

  If a nowait READUPDATE is executed, <count-read> has no meaning and can be omitted. The count of the number of bytes read is obtained when the I/O operation completes through the <count-transferred> parameter of the AWAITIO procedure.

  The READUPDATE procedure must complete with a corresponding call to the AWAITIO procedure when used with a file that is OPENed nowait.

- Default Locking Mode Action

  If the default locking mode is in effect when a call to READUPDATE is made to a locked file or record, but the <filenum> of the locked file differs from the <filenum> in the call, the caller of READUPDATE is suspended and queued in the "locking" queue behind other processes attempting to access the file or record.

  NOTE

  A deadlock condition occurs if a call to READUPDATE is made by the process having multiple openings on the same file, and the <filenum> used to lock the file differs from the <filenum> supplied to READUPDATE.

- Alternate Locking Mode Action

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

- Lock Mode by SETMODE

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

- Value of the Current Key and Current-Key Specifier

  For key-sequenced, relative, and entry-sequenced files, random processing implies that a designated record must exist. Therefore, positioning for READUPDATE is always to the record described by the exact value of the current key and current-key specifier. If such a record does not exist, the call to READUPDATE is rejected with a file system error 11 ("record does not exist"). This is unlike sequential processing through READ where positioning can be by approximate, generic, or exact key value.

Disc File Considerations

- Record Does Not Exist

  If the position specified for the READUPDATE operation does not exist, the call is rejected with error 11. (The positioning is specified by the exact value of the current key and current-key specifier).

- Structured Files

  --Calling READUPDATE without selecting a specific record

    If the call to READUPDATE immediately follows a call to KEYPOSITION where a nonunique alternate key is specified, the READUPDATE fails. A subsequent call to FILEINFO returns a file system error 46 ("invalid key"). However, if an intermediate call to READ or READLOCK is made, the call to READUPDATE is permitted because a unique record is identified.

  --Current-state indicators after READUPDATE

    After a successful READUPDATE, the current-state indicators are unchanged.

- Unstructured Disc Files

  --Reading unstructured files

    For a READ from an unstructured disc file, data transfer begins at the position indicated by the current-record pointer. A call to READUPDATE typically follows a call to POSITION that sets the current-record pointer to the desired relative byte address.

  --File pointer action for unstructured files is unaffected.

--Determination of <count-read> for unstructured files

After a successful call to READUPDATE to an unstructured file, the value returned in <count-read> is determined by:

<count-read> := $MIN(<read-count>,EOF - next-record pointer)

--The number of bytes read

If the unstructured file is created with the ODDUNSTR (odd unstructured file) parameter set, the number of bytes read is exactly that number specified with <read-count>. If the ODDUNSTR parameter is not set when the file is created, the value of <read-count> is rounded up to an even value before the READUPDATE is executed.

The ODDUNSTR parameter is set when the file is created, either with <filetype>.<12> of the CREATE procedure or with the Peripheral Utilities Program SET and CREATE commands.

- Even Unstructured Disc File

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

Interprocess Communication Considerations

- Replying to Messages

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

- Queuing Several Messages Before Replying

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

  If $RECEIVE is opened with <receive-depth> = 0, only READs can be performed, and READUPDATE and REPLY fail with error 2 ("operation not allowed on this type of file").

- Message Tags When Replying to Queued Messages

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

Example

```
CALL READUPDATE ( RECV^FNUM , REC^BUF , 512 );
```

Related Programming Manuals

For programming information about the READUPDATE file system procedure, refer to the GUARDIAN Operating System Programmer's Guide and the ENSCRIBE Programming Manual.

READUPDATELOCK PROCEDURE

The READUPDATELOCK procedure is used for random processing of records in a disc file. READUPDATELOCK locks, then reads the record from the current position in the file in the same manner as the combination of LOCKREC and READUPDATE. READUPDATELOCK is intended for reading a record after calling POSITION or KEYPOSITION, possibly in anticipation of a subsequent call to the WRITEUPDATE or WRITEUPDATEUNLOCK procedure.

A call to READUPDATELOCK is functionally equivalent to a call to LOCKREC followed by a call to READUPDATE. However, less system processing is incurred when one call is made to READUPDATELOCK rather than two separate calls to LOCKREC and READUPDATE.

The syntax for READUPDATELOCK is:

```
CALL READUPDATELOCK ( <filenum>                    ! i
                     ,<buffer>                     ! o
                     ,<read-count>                 ! i
                     ,[ <count-read> ]             ! o
                     ,[ <tag> ]  );                ! i
```

<filenum> input

INT:value

is the number of an open file that identifies the file to be read.

<buffer> output

INT:ref:*

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

<read-count> input

INT:value

is the number of bytes to be read {0:4096}.

→

<count-read> output

INT:ref:1

is for wait I/O only. <count-read> returns a count of the number of bytes returned from the file into <buffer>.

<tag> input

INT(32):value

is for nowait I/O only. <tag> is a value you define that uniquely identifies the operation associated with this READUPDATELOCK.

NOTE

The system stores the <tag> value until the I/O operation completes. The system then returns the <tag> information to the program in the <tag> parameter of the call to AWAITIO, thus indicating that the operation completed.

Condition Code Settings

< (CCL) indicates that an error occurred (call FILEINFO).

= (CCE) indicates that the READUPDATELOCK is successful.

> (CCG) does not return from READUPDATELOCK for disc files.

Considerations

- Nowait and READUPDATELOCK

  The READUPDATELOCK procedure must complete with a corresponding call to the AWAITIO procedure when used with a file that is OPENed nowait.

- If READUPDATELOCK is performed on nondisc files, an error is returned.

- Random Processing

  For key-sequenced, relative, and entry-sequenced files, random processing implies that a designated record must exist. Therefore, positioning for READUPDATELOCK is always to the record described by the exact value of the current key and current-key specifier. If such a record does not exist, the call to READUPDATELOCK is rejected with file system error 11.

- See "Disc File Considerations" for the READUPDATE procedure.

Example

```
CALL READUPDATELOCK ( IN^FILE , INBUFFER , 72 , NUM^READ );
```

Related Programming Manuals

For programming information about the READUPDATELOCK file system procedure, refer to the ENSCRIBE Programming Manual.

RECEIVEINFO PROCEDURE

The RECEIVEINFO procedure is used to obtain the process ID (PID), message tag, error recovery (sync ID), and request-related (file number and READ count) information associated with the last message read from the $RECEIVE file. This information is contained in the file's main-memory resident access control block (ACB); therefore, the application process is not suspended because of a call to RECEIVEINFO.

Note that the first two parameters to RECEIVEINFO (<process-id> and <message-tag>) duplicate the parameters to the LASTRECEIVE procedure.

NOTE

To avoid receiving invalid information from the $RECEIVE part of the ACB, call the RECEIVEINFO procedure immediately following the call to READUPDATE for $RECEIVE (or the AWAITIO that completes the READUPDATE). Do not perform another READUPDATE of $RECEIVE before calling RECEIVEINFO. However, you can check the condition code or call FILEINFO without changing the information in the ACB.

The syntax for RECEIVEINFO is:

```
CALL RECEIVEINFO ( [ <process-id> ]              ! o
                  ,[ <message-tag> ]             ! o
                  ,[ <sync-id> ]                 ! o
                  ,[ <filenum> ]                 ! o
                  ,[ <read-count> ] );           ! o
```

<process-id> output

INT:ref:4

returns the PID of the process that sent the last message read through the $RECEIVE file. If the process is in the destination control table (DCT), the information returned consists of:

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

→

If the process is not in the DCT, the information returned consists of:

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

<message-tag> output

INT:ref:1

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

<sync-id> output

INT(32):ref:1

returns the sync ID associated with this message.

<filenum> output

INT:ref:1

returns the file number of the file in the requesting process associated with this message.

<read-count> output

INT:ref:1

returns the number of bytes requested in reply to the message. If the message is the result of a request made in a call to WRITE, <read-count> will be 0. If the message is the result of a request made in a call to WRITEREAD, <read-count> is the same as the read count value passed by the requester to WRITEREAD.

Condition Code Settings

< (CCL) indicates that $RECEIVE is not open.

= (CCE) indicates that RECEIVEINFO is successful.

> (CCG) does not return from RECEIVEINFO.

Considerations

- PID and RECEIVEINFO

  The PID returned by RECEIVEINFO following receipt of an OPEN, CLOSE, CONTROL, SETMODE, RESETSYNC, or CONTROLBUF system message identifies the process associated with the operation.

- High-Order Three Words of PID Are Zero

  The high-order three words of the PID are zero following the receipt of system messages other than OPEN, CLOSE, CONTROL, SETMODE, RESETSYNC, and CONTROLBUF.

- Sync ID Definition

  A sync ID is a doubleword, unsigned integer. Each process file that is open 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. 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 or indirectly through the CHECKMONITOR procedure). Refer to the GUARDIAN Operating System Programmer's Guide for information about checkpointing.

  When a request is sent to a process (that is, CONTROL, CONTROLBUF, CLOSE, OPEN, SETMODE, WRITE, or WRITEREAD to a process file), the requestor'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 0.)

- Duplicate Requests

  The <sync-id> parameter allows the server process (that is, the 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.

NOTE

Neither a CANCELREQ or AWAITIO timeout completion have any affect on the sync ID (that is, it will be an ever-increasing value).

Also, the sync ID is independent of the <sync-depth> parameter to OPEN.

- Server Process Identifying Separate Opens by the Same Requester

  The <filenum> parameter allows the server process to identify separate OPENs by the same requester process. The value returned in <filenum> is the same as the file number used by the requester to make this request.

- Type of Request Being Made by the Requester

  The <read-count> parameter allows the server process to identify the type of request being made by the requester:

<read-count>	= 0	a WRITE request or WRITEREAD request with a <read-count> of zero is made.
<read-count>	<> = 0	or if <read-count> is greater, then the requester perform a WRITEREAD request of <read-count> bytes.

  This information can be used to determine if the requester sends data (for example, if <read-count> = 0, then requester is "listing") or also expects a reply (for example, if <read-count> > 0, then requester is "prompting").

Example

```
CALL RECEIVEINFO ( REQ^SYNCID , , , REQ^FNUM , REQ^READCOUNT );
```

Related Programming Manual

For programming information about the RECEIVEINFO file system procedure, refer to the GUARDIAN Operating System Programmer's Guide.

REFRESH PROCEDURE

The REFRESH procedure is used to write control information that passes to their associated physical disc volumes. REFRESH always writes out the control information contained in file control blocks (FCBs), such as end-of-file and free-list pointers. Optionally, the REFRESH procedure with the <all> parameter can be used to write out audit buffers and cache data buffers as well. Only the data and control information that is not already on disc are written out.

The REFRESH procedure should be used only when a volume is brought down (for example, immediately prior to a system cold load or PUP DOWN !).

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

The syntax for REFRESH is:

```
{ <error> := } REFRESH  ( [ <volname> ]                    ! i
{  CALL      }            ,[ <all> ]  );                   ! i
```

<error> returned value

INT

returns a file system error number indicating the outcome of the call. (Refer to the System Messages Manual for a list of all file system errors.)

<volname> input

INT:ref:4

specifies a volume whose associated FCBs should be written to disc. $<volname> can be specified as a full 12-word <filename>; <filename>[4:11] is ignored. <volname> can be either:

$<volname>
or
\<sysnum><volname>

If omitted, all FCBs for all volumes are written to their respective discs.

→

<all>	input
INT:value is one of the following values: 0 = WRITE out only dirty (used) FCBs. <> 0 = WRITE out dirty FCBs and dirty cached data blocks. 0 is the default if <all> is omitted.	

Condition Code Settings

The condition code has no meaning following a call to REFRESH.

Consideration

- When <volname> Is Omitted

  When REFRESH is called without a <volname>, the error return is always 0.

Example

```
ERROR := REFRESH;  ! refresh FCBs for all volumes.
```

Related Programming Manuals

For programming information about the REFRESH file system procedure, refer to the ENSCRIBE Programming Manual.

REMOTEPROCESSORSTATUS PROCEDURE

The REMOTEPROCESSORSTATUS procedure supplies the status of processor modules in a particular system in a network.

The syntax for REMOTEPROCESSORSTATUS is:

```
<status> := REMOTEPROCESSORSTATUS ( <sysnum> );        ! i
```

<status> returned value

INT(32)

returns two words indicating the processor status.

The most significant word (MSW) is the number of processors in the remote system. The least significant word (LSW) is a bit mask for processor availability.

Word [0]	MSW, the remote system is nonexistent or unavailable
Word [1]	LSW, bit mask: 1 = the processor is up 0 = the processor is down or nonexistent

<sysnum> input

INT:value

is the number of a particular system in a network whose processor modules' status is returned.

Condition Code Settings

The condition code has no meaning following a call to REMOTEPROCESSORSTATUS.

Considerations

- Where to Find the System Number

  The system number for a particular system whose name is known can be obtained from the LOCATESYSTEM procedure.

- Equivalencing the Two Words of <status>

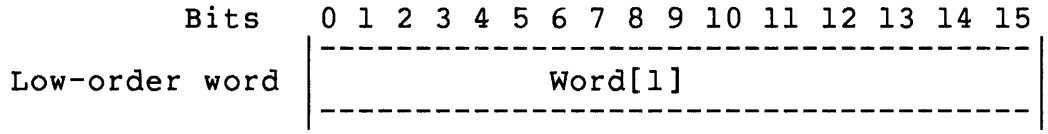
  The two words of <status> can be separated by the usual technique of equivalencing INT variables to the high- and low-order words. For example, a Tandem Application Language procedure that calls REMOTEPROCESSORSTATUS might contain the following declarations:

```
INT(32) STATUS;
INT NUM^PROCESSORS = STATUS;         ! high-order word.
INT BIT^MASK = NUM^PROCESSORS + 1;   ! low-order word.
```

  See the Transaction Application Language (TAL) Reference Manual. for an explanation of equivalenced variables

- Low-Order Word of <status>

  The bits in the low-order word are ordered from 0 to 15, from left to right (the CPU number corresponds to the bit number):



- Using <status> for Local Processors

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

```
INT(32) MY^PROCESSOR^STATUS;
MY^PROCESSOR^STATUS := REMOTEPROCESSORSTATUS( MYSYSTEMNUMBER );
```

Example

```
STAT := REMOTEPROCESSORSTATUS ( SYS^NUM );
```

Related Programming Manual

None

REMOTETOSVERSION PROCEDURE

The REMOTETOSVERSION procedure provides an identifying letter and number indicating which version of the GUARDIAN operating system is running on a remote system.

The syntax for REMOTETOSVERSION is:

```
<tos-version> := REMOTETOSVERSION [ ( <sysid> ) ];        ! i


<tos-version>                  returned value

  INT

  returns a value of the form:

    <0:7>   uppercase ASCII letter indicating system level:
             A    = TOS
             B    = GUARDIAN
             C    = GUARDIAN / 1.1
             D    = GUARDIAN / EXPAND
             E    = GUARDIAN / EXPAND / Transaction Monitoring
                    Facility
             K,L  = GUARDIAN, NonStop system

    <8:15>   revision number of system in binary

    Zero is returned if the system <sysid> does not exist or is
    inaccessible.


<sysid>                        input

  INT:value

  is the system number that identifies the system for which
  the operating system version is returned.  If <sysid> is
  omitted, it defaults to the local system.
```

Condition Code Settings

The condition code has no meaning following a call to REMOTETOSVERSION.

Example

```
REMOTE^VERSION := REMOTETOSVERSION ( SYSTEM^NUM );
```

Related Programming Manual

For programming information about the REMOTETOSVERSION system procedure, refer to the GUARDIAN Operating System Programmer's Guide.

RENAME PROCEDURE

The RENAME procedure is used to change the name of a disc file that is open. If the file is temporary, assigning a name causes the file to be made permanent.

A call to the RENAME procedure is rejected with an error indication if there are incomplete nowait operations pending on the specified file.

The syntax for RENAME is:

```
CALL RENAME ( <filenum>                    ! i
            ,<new-name> );                 ! i


<filenum>                      input

  INT:value

  is the number of an open file that identifies the file to be
  renamed.


<new-name>                     input

  INT:ref:12

  is an array containing the file name to be assigned to the disc
  file, as follows:

    <filename>[0:3]  =  $<volname><blank-fill>
                                or
                        \<sysnum><volname><blank-fill>
              [4:7]  =  <subvolname><blank-fill>
              [8:11] =  <disc filename><blank-fill>
```

Condition Code Settings

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

Considerations

- Purge Access for RENAME

  The caller must have purge access to the file for the RENAME to be successful. Otherwise, the RENAME is rejected with file system error 48, "security violation."

- Volume Specification for <new-name>

  The volume specified in <new-name> must be the same as the volume specified when opening the file. Neither the volume name nor the system name can be changed by RENAME.

- <sysnum> Specification for <new-name>

  If <sysnum> is specified as part of <new-name>, it must be the same as the system number used when the file was initially opened.

- Partitioned Files

  When the primary partition of a partitioned file is renamed, the file system automatically renames all other partitions located anywhere in the network.

- Renaming a File Audited by the Transaction Monitoring Facility (TMF)

  The file to be renamed cannot be a file audited by TMF. An attempt to rename such a file fails with file system error 80 ("invalid operation attempted on audited file or nonaudited disc volume (device type 3)").

- Structured Files With Alternate Keys

  If the primary-key file is renamed, it is linked with the alternate-key file. If you rename the alternate-key file and then try to access the primary-key file, file system error 4 occurs, because the primary-key file is still linked with the old name for the alternate-key file. You can use the FUP ALTER command to correct this problem.

Example

```
CALL RENAME ( TEMP^FILENUM , NEW^NAME );
```

Related Programming Manual

For programming information about the RENAME file system procedure, refer to the ENSCRIBE Programming Manual.

REPLY PROCEDURE

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

The REPLY procedure can be called even if there are incomplete nowait I/O operations pending on $RECEIVE.

The syntax for REPLY is:

```
CALL REPLY ( [ <buffer> ]                          ! i
           ,[ <write-count> ]                      ! i
           ,[ <count-written> ]                    ! o
           ,[ <message-tag> ]                      ! i
           ,[ <error-return> ] );                  ! i
```

<buffer> input

INT:ref:*

is an array containing the reply message.

<write-count> input

INT:value

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

<count-written> output

INT:ref:1

returns a count of the number of bytes written to the file.

<message-tag> input

INT:value

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

<error-return> input

INT:value

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

	File System Error	Condition Code Setting
<error-return>	10-511	CCL (error)
<error-return>	0	CCE (no error)
<error-return>	1-9	CCG (warning)

NOTE

Error numbers 300-511 are reserved for user applications; errors numbers 10-255 are Tandem errors.

The <error-return> value returns in the <error> parameter of FILEINFO when the originator calls FILEINFO for the associated completion.

If <error-return> is omitted, a no-error indication (that is, 0) returns to the message originator.

Condition Code Settings

< (CCL) indicates that an error occurred (call FILEINFO).

= (CCE) indicates the REPLY is successful.

> (CCG) does not return from REPLY.

Considerations

- Replying to Queued Messages

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

  If $RECEIVE is opened with <receive-depth> = 0, only READs can be performed; READUPDATE and REPLY fail with error 2 ("operation not allowed on this type of file").

- Using the <message-tag>

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

- Error Handling

  The <error-return> parameter can be used to return an error indication to the requester in response to the OPEN, CONTROL, SETMODE, and CONTROLBUF system messages. The error returns to the requester when the associated I/O procedure completes.

Example

```
CALL REPLY ( OUT^BUFFER , 512 );
```

Related Programming Manual

For programming information about the REPLY file system procedure, refer to the GUARDIAN Operating System Programmer's Guide.

REPOSITION PROCEDURE

The REPOSITION procedure is used to position a disc file to a saved position (the positioning information having been saved by a call to the SAVEPOSITION procedure).

Following a call to REPOSITION, the disc file is positioned to the point where it was when SAVEPOSITION was called. That is, the READ that took place prior to calling SAVEPOSITION resulted in an update of the next-record pointer; this pointer is used in the next READ after calling REPOSITION.

A call to the REPOSITION procedure is rejected with an error indication if any incomplete nowait operations are pending on the specified file.

The syntax for REPOSITION is:

```
CALL REPOSITION ( <filenum>                          ! i
                 ,<positioning-block> );             ! i
```

<filenum> input

INT:value

is the number of an open file that identifies the file to be positioned to a saved position.

<positioning-block> input

INT:ref:*

indicates a saved position to be repositioned to.

Condition Code Settings

< (CCL) indicates that an error occurred (call FILEINFO).

= (CCE) indicates that REPOSITION is successful.

> (CCG) indicates that the file is not a disc file.

Example

```
CALL REPOSITION ( FILE^NUM , POSITION^BLOCK );
```

Related Programming Manual

For programming information about the REPOSITION file system procedure, refer to the ENSCRIBE Programming Manual.

RESERVELCBS PROCEDURE

The RESERVELCBS procedure allows you to reserve link control blocks (LCBs). You can specify the number of LCBs to be reserved for receiving messages (that is, receive LCBs) and the number of LCBs to be reserved for sending messages (that is, send LCBs).

The syntax for RESERVELCBS is:

```
CALL RESERVELCBS ( <no-receive-lcbs>                    ! i
                  ,<no-send-lcbs>  );                   ! i
```

<no-receive-lcbs> input

INT:value

is the number of receive LCBs to be reserved for this process: {0:255}.

<no-send-lcbs> input

INT:value

is the number of send LCBs to be reserved for this process: {0:255}.

Condition Code Settings

>	(CCL)	indicates that not enough unreserved LCBs are available to reserve the LCBs specified in this call, or the amount requested by either parameter is not in the range of {0:255}. The number of reserved LCBs allocated to this process is unchanged.
=	(CCE)	indicates that the requested LCBs are reserved for this process.
<	(CCG)	is not returned by RESERVELCBS.

Considerations

- A process can call RESERVELCBS multiple times; the latest call is the one that is used.

- The worst-case values for <no-receive-lcbs> and <no-send-lcbs> can be calculated as follows:

  <no-receive-lcbs> = 1 for each terminal where BREAK is being monitored
  +1 for each CPU being monitored (that is, by MONITORCPUS)
  +1 for each process of which this process is the creator or ancestor
  +1 for each possible interprocess message from application processes.

  NOTE

  You should try to reduce the above worst-case value of <no-receive-lcbs>. This value can be safely reduced by assuming that only some of the indicated messages will be queued at any one time. For example, normally only one CPU DOWN message will be pending at any given time.

  <no-send-lcbs> = 1 for each nowait operation that can be in progress at any given moment
  +1 for a wait I/O operation or call to a process control procedure.

- If a process runs out of its reserved LCBs, the system will attempt to secure the needed LCBs from the LCB pool.

- If a process has reserved "send" and "receive" LCBs, console message 52 (RECEIVE QUEUE FOR ... HAS MORE THAN 10 REQUESTS) will not be displayed, even if it has more than 10 requests queued on its $RECEIVE file. (If LCBs are not reserved, message 52 is displayed only when the count of allocated receive LCBs goes from 10 to 11.)

Example

```
CALL RESERVELCBS ( 1 , 4 );
```

Related Programming Manual

For programming information concerning the RESERVELCBS procedure, refer to the GUARDIAN Operating System Programmer's Guide.

RESETSYNC PROCEDURE

The RESETSYNC procedure is used by the backup process of a process pair after a failure of the primary process. With this procedure, a different series of operations are performed from those performed by the primary before its failure. The RESETSYNC procedure does the following:

- Clears a process pair's file synchronization block so that the operations performed by the backup are not erroneously related to the operations just completed by the primary process
- Resynchronizes any open files whose file sychronization blocks are not checkpointed after the most recent stack checkpoint.

NOTE

Typically, RESETSYNC is not called directly by application programs. Instead, it is called indirectly by CHECKMONITOR.

The syntax for RESETSYNC is:

```
CALL RESETSYNC ( <filenum> );          ! i

<filenum>                    input

  INT:value

  is the number of an open file that identifies the file whose
  synchronization block is to be cleared.
```

Condition Code Settings

< (CCL) indicates that an error occurred (call FILEINFO).

= (CCE) indicates that RESETSYNC is successful.

> (CCG) indicates that the file is not a disc file.

Considerations

- File Number Has Not Been Opened

  If the RESETSYNC file number does not match the file number of the open file that you are trying to access, the call to RESETSYNC is rejected with file system error 16.

- Not Receiving Messages

  If the $RECEIVE file is not opened with <flags>.<1> set to one to enable receipt of RESETSYNC messages, the procedure fails with file system error 7.

Example

```
CALL RESETSYNC( FILE^NUMBER );
```

Related Programming Manual

For programming information about the RESETSYNC checkpointing procedure, refer to the GUARDIAN Operating System Programmer's Guide.

RESUMETRANSACTION PROCEDURE

RESUMETRANSACTION restores to currency a TMF transaction identifier created by a previous call to BEGINTRANSACTION.

RESUMETRANSACTION permits requesters to be multithreaded, that is, to concurrently process two or more transactions. At any instant, the requester can only work on one transaction, but the requester can switch to any transaction that it previously began by calling RESUMETRANSACTION.

This procedure is called with the transaction tag returned by the call to BEGINTRANSACTION. The transaction identifier associated with the tag becomes the current-transaction identifier for the process calling RESUMETRANSACTION.

The syntax for RESUMETRANSACTION is:

```
<status> := RESUMETRANSACTION ( <trans-begin-tag> );     ! i
```

<status> returned value

INT

returns a 0, if successful or a file system error number. (Refer to the System Messages Manual for a list of all file system errors.)

<trans-begin-tag> input

INT(32):value

is the value returned by the optional <trans-begin-tag> parameter of BEGINTRANSACTION. If the value of this parameter is 0D, the current-transaction identifier of the calling process is reset to 0, indicating no transaction identifier.

Condition Code Settings

The condition code has no meaning following a call to RESUMETRANSACTION.

Considerations

- The Current-Transaction Identifier

  If the transaction identifier identified by <trans-begin-tag> is begun by the calling process or its backup, it becomes the current-transaction identifier for the calling process even if an error returns.

- RESUMETRANSACTION does not change the current-transaction identifier for the backup of the calling process.

- Invalid or Obsolete Transaction Identifier

  If the transaction is not begun by the process that issued BEGINTRANSACTION (or its backup), or the transaction is no longer in the system, this call returns file system error 78.

- Transaction Aborts

  If the parent process (BEGINTRANSACTION process) of this transaction fails, this call is rejected and returns file system error 90.

- Path in Network Node Is Down

  If the path to a participating network node is down, the transaction aborts, and the call returns file system error 92.

- Spanning Too Many Audit-Trail Files

  If the transaction tried to span too many audit-trail files the transaction is aborted, and a subsequent call to RESUMETRANSACTION returns file system error 93.

- Operator Command Response

  If an operator aborts the transaction through the TMFCOM "ABORT TRANSACTION" command, the system aborts the transaction, and the call returns file system error 94.

- Calling ABORTTRANSACTION Before ENDTRANSACTION

  If a previous call to ABORTTRANSACTION is made before the call to RESUMETRANSACTION, the call aborts, and returns file system error 97.

Example

```
STATUS := RESUMETRANSACTION ( TRANS^BEGIN^ID );
```

Related Programming Manual

For programming information about the RESUMETRANSACTION procedure, refer to the Transaction Monitoring Facility (TMF) Reference Manual.

SAVEPOSITION PROCEDURE

The SAVEPOSITION procedure is used to save a disc file's current file positioning information in anticipation of a need to return to that position. The positioning information is returned to the file system in a call to the REPOSITION procedure when you want to return to the saved position.

The syntax for SAVEPOSITION is:

```
CALL SAVEPOSITION ( <filenum>                              ! i
                  ,<positioning-block>                     ! o
                  ,[ <positioning-blksize> ] );            ! o
```

<filenum> input

INT:value

is a number of an open file, identifying the file whose positioning block is to be obtained.

<positioning-block> output

INT:ref:*

returns the positioning block for this file's current position.

<positioning-blksize> output

INT:ref:1

returns the number of words in the positioning block.

For key-sequenced files where positioning is performed by:

- Primary key, the count is calculated by

  7 + (<primary-key> + 1) / 2

- An alternate key, the count is calculated by

  7 + (<max-alternate-keylen> + <primary-keylen> + 1) / 2

For unstructured files and for entry-sequenced and relative files where no alternate keys exist, the count is 4.

→

For entry-sequenced and relative files where positioning is performed by:

- Primary key, the count is 7
- An alternate key, the count is calculated by

  7 + (<max-alternate-keylen> + 4 + 1) / 2

Condition Code Settings

< (CCL) indicates that an error occurred (call FILEINFO).

= (CCE) indicates that SAVEPOSITION is successful.

> (CCG) indicates that the file is not a disc file.

Example

```
CALL SAVEPOSITION ( FILE^NUM , POSITION^BLOCK );
```

Related Programming Manual

For programming information about the SAVEPOSITION file system procedure, refer to the ENSCRIBE Programming Manual.

SETLOOPTIMER PROCEDURE

The SETLOOPTIMER procedure has two uses:

1. To abort the caller if the caller begins looping (malfunctioning)

2. To permit the caller to calculate the amount of processor time it has used

A call to the SETLOOPTIMER procedure is used to set the caller's "process loop-timer" value. A positive loop-timer value enables process loop timing by the operating system and specifies a limit on the total amount of processor time the calling process is allowed. If loop timing is enabled, the operating system decrements the loop-timer value as the process executes (that is, is in the active state). If the loop timer is decremented to 0 (indicating that the time limit is reached), a "process loop-timer timeout" trap occurs (trap number 4). Loop timing is disabled by specifying a loop-timer value of 0.

The syntax for SETLOOPTIMER is:

```
CALL SETLOOPTIMER ( <new-time-limit>                    ! i
                  ,[ <old-time-limit> ] );              ! o
```

<new-time-limit> input

INT:value

specifies the new time-limit value, in .01-second units, to be set into the process's loop timer. <new-time-limit> must be a positive value.

If 0 is passed as the <new-time-limit> value, process loop timing is disabled.

<old-time-limit> output

INT:ref:1

returns the current setting of the process's loop timer (in .01-second units).

Condition Code Settings

< (CCL) indicates that the <new-time-limit> parameter is omitted or is specified as a negative value. The state of process loop timing and the setting of the process's loop timer are unchanged.

= (CCE) indicates that the <new-time-limit> value is set into the process's loop timer and that loop timing is enabled.

> (CCG) does not return from SETLOOPTIMER.

Considerations

- SETLOOPTIMER is not practical for generating timed asynchronous interrupts for most users.

- Process Loop Timeout in System Code

  In operating system versions prior to the B00 version, if any trap occurs in the system code, control is given immediately to the user trap handler if one is defined, or otherwise to DEBUG or INSPECT. The process then abends. The stack markers are cut back to the first marker in user code (user library) segments. To the trap handler (or DEBUG) it appears as if the trap occurred in the environment of this marker (refer to the System Description Manual for more details).

  In the B00 version, this situation changes for the process loop-timeout trap. If a process loop-timeout trap occurs in system code, the trap is taken the next time control enters user code; the trap is delayed until then.

- Detection of Process Looping

  To detect itself looping, a process calls SETLOOPTIMER (resetting the time limit) at a given point each time through its main execution loop. If the process fails to execute through its main loop, SETLOOPTIMER is not called and the time limit is not reset. Consequently, the time limit is reached, and a trap occurs. (When the trap handler completes execution, the process resumes its normal instruction path.)

For example, a process's main execution loop can be written as follows:

```
-->
|    |
|  start:
|    CALL SETLOOPTIMER ( 1000 );
|    IF < THEN ... ;
|      |  .
|      |  . ! enable loop timing.  Time-limit value is 10 seconds.
|      |  .
|    CALL WRITEREAD ( termfnum ,... );
|      |  .
|      |  . ! process executes when terminal input is made.
|      |  . ! Loop-timer value is not decremented while process is
|      |  . ! suspended waiting for I/O.
|      |
|    terminal input is processed.
-----
```

- Process Timing

SETLOOPTIMER can be used to measure the CPU time to execute a set of instructions. For example,

```
    .
    .
CALL SETLOOPTIMER ( OLDVAL );  ! start timing.
    .
    .
  ! do the computation.
    .
    .
CALL SETLOOPTIMER ( 0, NEWVAL );  ! disable timing.
TIME^FOR^COMPUTATION := OLDVAL-NEWVAL);
    .
    .
    .
```

The following considerations apply:

--OLDVAL should be larger than the time estimated for the computation or else a loop-timeout trap will occur in the middle of the computation that is being measured.

--There is computational overhead associated with a call to the SETLOOPTIMER procedure and TIME^FOR^COMPUTATION can be inflated by up to 10ms because of this overhead. Therefore it is recommended that this use of the SETLOOPTIMER procedure be made for large computations only.

Example

```
CALL SETLOOPTIMER ( NEW^TIME );
```

Related Programming Manual

For programming information about the SETLOOPTIMER process control procedure, refer to the GUARDIAN Operating System Programmer's Guide.

SETMODE PROCEDURE

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

A call to the SETMODE procedure is rejected with an error indication if incomplete nowait operations are pending on the specified file.

The syntax for SETMODE is:

```
CALL SETMODE ( <filenum>                          ! i
             ,<function>                          ! i
             ,[ <param1> ]                        ! i
             ,[ <param2> ]                        ! i
             ,[ <last-params> ]  );               ! o
```

<filenum> input

INT:value

is a number of an open file that identifies the file to receive the SETMODE <function>.

<function> input

INT:value

is one of the device-dependent functions listed in the SETMODE functions table (see Appendix C).

<param1> input

INT:value

is one of the parameters listed in the SETMODE functions table (see Appendix C). If omitted, for a disc file the present value is retained. For SETMODEs on other devices, this value depends on the device and the value supplied in the <function> parameter.

⟶

<param2> input

INT:value

is one of the parameters listed in the SETMODE functions table (see Appendix C). If omitted, for a disc file the present value is retained. For SETMODEs on other devices this value depends on the device and the value supplied in the <function> parameter.

<last-params> output

INT:ref:2

returns the previous settings of <param1> and <param2> associated with the current <function>. The format is:

<last-params>[0] = old <param1>
<last-params>[1] = old <param2> (if applicable)

Condition Code Settings

< (CCL) indicates that an error occurred (call FILEINFO).

= (CCE) indicates that the SETMODE is successful.

> (CCG) indicates that the SETMODE function is not allowed for this device type.

Considerations

- Default SETMODE Settings

  The SETMODE settings designated as "default" are the values that apply when a file is opened (not if a particular <function> is omitted when SETMODE is called).

- Waited SETMODE

  The SETMODE procedure is used on a file as a waited operation even if <filenum> has been opened for nowait. Use the SETMODENOWAIT procedure for nowait operations.

Disc File Consideration

- Ownership and Security of File

  "Set disc file security" and "set disc file owner" are rejected unless the requester is the owner of the file or the super ID.

Interprocess Communication Considerations

- Nonstandard Parameter Values

  Any value can be specified for the <function>, <param1>, and <param2> parameters. An application-defined protocol should be established for interpreting nonstandard parameter values.

- Omitting <last-params>

  The <last-params> parameter must be omitted for interprocess files.

Messages

- Process SETMODE

  The issuance of a SETMODE to a file representing another process can cause a SETMODE system message (-33) to be sent to that process.

  The process ID of the caller to SETMODE can be obtained in a subsequent call to LASTRECEIVE or RECEIVEINFO. (Refer to Appendix F for a list of all system messages sent to processes.)

Examples

1.
```
LITERAL SECURITY = %0222;
   .
   .
   .
CALL SETMODE ( FNUM , 1 , SECURITY );
```

The LITERAL above sets the file's security to:

```
read    = any local user
write   = owner only
execute = owner only
purge   = owner only
```

```
2.      LITERAL PROG^SEC = %102202;
           .
           .
           .
        CALL SETMODE ( PFNUM , 1 , PROG^SEC );
```

The second LITERAL specifies that the file's owner ID should be used as the process's accessor ID when the program file is run. This is done by setting the file's security to:

```
set accessor ID to owner's ID when file is run
read    = owner only
write   = owner only
execute = any local user
purge   = owner only
```

Related Programming Manuals

For programming information about the SETMODE file system procedure, refer to the GUARDIAN Operating System Programmer's Guide, the ENSCRIBE Programming Manual, and the data-communication manuals.

SETMODENOWAIT PROCEDURE

The SETMODENOWAIT procedure is used to set device-dependent functions in a nowait manner on nowait files.

Whereas the SETMODE procedure is a waited operation and suspends the caller while waiting for a request to complete, the SETMODENOWAIT procedure returns to the caller after initiating a request. A call to SETMODENOWAIT completes in a call to AWAITIO. The <count-transferred> parameter to AWAITIO has no meaning for SETMODENOWAIT completions. The <buffer-addr> parameter is set to the address of <last-params> parameter of SETMODENOWAIT.

The syntax for SETMODENOWAIT is:

```
CALL SETMODENOWAIT ( <filenum>                    ! i
                   ,<function>                    ! i
                   ,[ <param1> ]                  ! i
                   ,[ <param2> ]                  ! i
                   ,[ <last-params> ]             ! o
                   ,[ <tag>]  );                  ! i
```

<filenum> input

INT:value

is a number of an open file, identifying the file to receive the SETMODENOWAIT <function>.

<function> input

INT:value

is one of the device-dependent functions listed in the SETMODE functions table (see Appendix C).

<param1> input

INT:value

is one of the <param1> values listed in the SETMODE functions table (see Appendix C). If omitted, for a disc file the present value is retained. For SETMODEs on other devices, this value depends on the device and the value supplied in the <function> parameter.

→

<param2> input

INT:value

is one of the <param2> values listed in the SETMODE functions table (see Appendix C). If omitted, for a disc file the present value is retained. For SETMODEs on other devices this value depends on the device and the value supplied in the <function> parameter.

<last-params> output

INT:ref:2

returns the previous settings of <param1> and <param2> associated with the current <function>. The format is:

```
<last-params>[0] = old <param1>
<last-params>[1] = old <param2> (if applicable)
```

<tag> input

INT(32):value

is for nowait I/O only. <tag> is a value you define that uniquely identifies the operation associated with this SETMODENOWAIT.

NOTE

The system stores the <tag> value until the I/O operation completes. The system then returns the <tag> information to the program in the <tag> parameter of the call to AWAITIO, thus indicating that the operation completed.

Condition Code Settings

< (CCL) indicates that an error occurred (call FILEINFO).

= (CCE) indicates that the SETMODENOWAIT is successful.

> (CCG) indicates that the SETMODENOWAIT function is not allowed for this device type.

Considerations

- File Opened With Wait Depth > 0

  AWAITIO must be called to complete the call when <filenum> is opened with a wait depth greater than 0. For files with wait depth equal to 0, a call to SETMODENOWAIT is a waited operation and performs in the same way as a call to SETMODE.

- <last-params> and AWAITIO

  The <buffer> parameter of AWAITIO is set to @<last params>, and the count is undefined.

Example

```
LITERAL SET^SPACE = 6,
        NO^SPACE = 0,
        SPACE = 1;
          .
          .
CALL SETMODENOWAIT ( FILE^NUM , SET^SPACE , SPACE );
     ! turns off single spacing for a line printer.
```

Related Programming Manuals

For programming information about the SETMODENOWAIT file system procedure, refer to the GUARDIAN Operating System Programmer's Guide, the ENSCRIBE Programming Manual, and the data communication manuals.

SETMYTERM PROCEDURE

The SETMYTERM procedure permits a process to change the terminal that is used as its home terminal (the default home terminal is the home terminal of a process's creator).

The syntax for SETMYTERM is:

```
CALL SETMYTERM ( <terminal-name> );                    ! i

<terminal-name>              input

  INT:ref:12

  contains the file name of the terminal or the process that is to
  function as the caller's home terminal.
```

Condition Code Settings

< (CCL) indicates that the terminal cannot be reassigned, <terminal-name> is invalid, or <terminal-name> is not a terminal or a named process.

= (CCE) indicates that the SETMYTERM is successful.

> (CCG) does not return from SETMYTERM.

Considerations

- New Processes Started After SETMYTERM Is Called

  If the caller to SETMYTERM creates any processes after the call to SETMYTERM, the new home terminal is the home terminal for those processes. SETMYTERM has no effect on any existing process created by the caller.

Example

```
CALL SETMYTERM ( TERM );
```

Related Programming Manual

For programming information about the SETMYTERM process control procedure, refer to the GUARDIAN Operating System Programmer's Guide.

SETPARAM PROCEDURE

The SETPARAM procedure is used to set and fetch various values such as the station characteristics of network addresses.

The syntax for SETPARAM is:

```
CALL SETPARAM ( <filenum>                                  ! i
              ,<function>                                  ! i
              ,[ <param-array> ]                           ! i
              ,[ <param-count> ]                           ! i
              ,[ <last-param-array> ]                      ! o
              ,[ <last-param-count> ] );                   ! o
```

<filenum> input

INT:value

is the number of an open file for which special information is sent.

<function> input

INT:value

is one of the following SETPARAM function codes:

1 = set or fetch a remote data terminal equipment address (use with X.25 Access Method (X25AM) only)

2 = set or fetch the clear cause or diagnostic bytes (use with X25AM only)

3 = set or fetch parameters for BREAK handling

4 = set or fetch the reset cause or diagnostic bytes (use with X25AM only)

5 = fetch the restart cause or diagnostic bytes (use with X25AM only)

6 = set or fetch the 6520 and 6530 block mode terminal error counters (use with interactive terminal interface (ITI) protocol in X25AM only)

→

7 = set or override the closed user's group (CUG) number to be used in next call request packet

8 = set or fetch the protocol ID field in the outgoing call request packet (use with process-to-process protocol in X25AM only)

9 = fetch the reason why circuit disconnected and learn the current link status (use with X25AM only)

20 = reset and retrieve the called data terminal equipment (DTE) address buffer

21 = provide a count of the number of 64-byte segments that can be sent and received by a subdevice

22 = access the Level 4 ITI protocol block mode timer

For a detailed description of function 3, refer to the Device-Specific Access Methods--(AM3270/TR3271) and the Device-Specific Access Method--(AM6520) manuals. For a detailed description of functions 1, 2, 4, 5, 6, 8, and 9, refer to the X.25 Access Method--(X25AM) manual.

<param-array> input

INT:ref:*

is a list or string as required by <function>.

<param-count> input

INT:value

is the number of bytes contained in <param-array>.

<last-param-array> output

INT:ref:*

returns previous parameter settings associated with <function>.

→

<last-param-count> output
INT:ref:1
returns the length of <last-param-array> in bytes (maximum of 256 bytes).

Condition Code Settings

The condition code has no meaning following a call to SETPARAM.

Considerations

<param-array> and <last-param-array>

These are integer arrays containing:

```
word   [0] = equivalent to parameter 1 of SETMODE 11
       [1] = equivalent to parameter 2 of SETMODE 11
       [2] = most significant word of the break tag
       [3] = least significant word of the break tag
```

where:

[0] = 0, disable BREAK (default setting)
= <cpu,pin>, enable BREAK

[1] (Terminal access mode after BREAK is typed.)
= 0, normal mode (any file-type access is permitted)
= 1, break mode (only break-type file access permitted).

[2:3] = are the two words of the 32-bit tag. This is saved by the I/O process handling the 3270 terminal. Whenever AM3270 detects the input of a PA1 key (simulated BREAK) from a 3270 terminal configured for ITI protocol, a BREAK message line control block (LCB) is sent to the owner of the BREAK for that terminal. (The owner of BREAK is specified by the parameter in <buf>[0].) The six parameter words of the BREAK message are:

P1 = -20 (identifies this as a BREAK message)
P2 = logical device number of the I/O process
P3 = system number of the I/O process
P4 = most significant word of the BREAK tag
P5 = least significant word of the BREAK tag
P6 = 0

To file system users, the BREAK message LCB parameters P1 through P6 appear in a buffer READ from $RECEIVE as buffer[0:5].

The break tag is checkpointed to its backup by the I/O process (to satisfy NonStop requirements).

Example

```
CALL SETPARAM ( F , 8 , , , OLD^PARAMS , OLD^SIZE );
```

The above example fetches the protocol ID field in the outgoing call request packet. Four bytes of data return.

Related Programming Manuals

For programming information about the SETPARAM procedure, refer to the data-communication manuals.

SETSTOP PROCEDURE

The SETSTOP procedure permits a process to protect itself from being deleted by any process other than itself or its creator.

The syntax for SETSTOP is:

```
{ <last-stop-mode> := } SETSTOP ( <stop-mode> );        ! i
{ CALL               }
```

<last-stop-mode> returned value

INT

returns a value that is either the preceding value of <stop-mode> or -1 if an illegal mode was specified.

<stop-mode> input

INT:value

is a value specifying a new stop mode. The modes are:

- 0 = stoppable by any process
- 1 = stoppable only by:
  - a. the super ID
  - b. a process whose process accessor ID = this process's creator ID
  - c. a process whose process accessor ID = this process's accessor ID (this includes the caller to STEPMOM)
- 2 = unstoppable; this mode available to privileged users only

Refer to the GUARDIAN Operating System Programmer's Guide for additional information about super ID or process accessor ID.

Condition Code Settings

The condition code has no meaning following a call to SETSTOP.

Considerations

- The default stop mode is 1 when a process is created.
- Setting a Process's Stop Mode to 1 and Then Issuing a Stop

  If a process's stop mode is 1 and a STOP is issued to it by a process without the authority to stop it, the process does not stop; the process is deleted, however, if and when the stop mode is changed to 0.
- Setting a Process's Stop Mode to 2 and Then Issuing a Stop

  If a process's stop mode is 2, and a STOP is issued to it by another process, stop is queued until the process is in a stoppable mode.

Example

```
LAST^MODE := SETSTOP ( NEW^MODE );
```

Related Programming Manual

For programming information about the SETSTOP process control procedure, refer to the GUARDIAN Operating System Programmer's Guide.

SETSYNCINFO PROCEDURE

The SETSYNCINFO procedure is used by the backup process of a process pair after a failure of the primary process.

The SETSYNCINFO procedure passes a process pair's latest synchronization block (received in a checkpoint message from the primary) to the file system. Following a call to the SETSYNCINFO procedure, the backup process can retry the same series of WRITE operations started by the primary before its failure. The use of the sync block ensures that operations which might have been completed by the primary before its failure are not duplicated by the backup.

NOTE

Typically, SETSYNCINFO is not called directly by application programs. Instead, it is called indirectly by CHECKMONITOR.

The syntax for SETSYNCINFO is:

```
CALL SETSYNCINFO ( <filenum>                          ! i
                 ,<sync-block> );                     ! o
```

<filenum> input

INT:value

is the number of an open file that identifies the file whose synchronization block is being passed.

<sync-block> output

INT:ref:*

is the latest synchronization block received from the primary process.

Condition Code Settings

< (CCL) indicates that an error occurred (call FILEINFO).

= (CCE) indicates that SETSYNCINFO is successful.

> (CCG) indicates that the file is not a disc file.

Considerations

- File Number Has Not Been Opened

  If the SETSYNCINFO file number does not match the file number of the open file, then the call to SETSYNCINFO is rejected with file system error 16.

- Application Parameter or Buffer Address Out of Bounds

  If an out-of-bounds application buffer address parameter is specified in the SETSYNCINFO call (that is, a pointer to the buffer has an address that is greater than the MEM associated with the data area of the process), then the call is rejected with file system error 22.

- Checksum Error on File Sync Block

  If an attempt is made to modify the file system sync buffer area, the SETSYNCINFO call is rejected with file system error 41.

Example

```
CALL SETSYNCINFO ( F1 , SYNC );
```

Related Programming Manual

For programming information about the SETSYNCINFO checkpointing procedure, refer to the GUARDIAN Operating System Programmer's Guide.

SETSYSTEMCLOCK PROCEDURE

The SETSYSTEMCLOCK procedure allows you to change the system clock if you are executing as a super group ID.

The syntax for SETSYSTEMCLOCK is:

```
CALL SETSYSTEMCLOCK ( <julian-gmt>                    ! i
                    , <mode>                          ! i
                    ,[ <tuid>]  );                    ! i
```

<julian-gmt> input

FIXED

is the Julian timestamp.

<mode> input

INT:value

specifies the mode and source as follows:

Mode	Source
0 = absolute Greenwich mean time (GMT)	operator input
1 = absolute GMT	hardware clock
2 = relative GMT	operator input
3 = relative GMT	hardware clock

The relative mode implies that the <julian-gmt> parameter contains a microsecond-based error correction factor, not an actual timestamp. This is used for precise time synchronization to a hardware clock or for a moderately precise method of operator time adjustment.

<tuid> input

INT:value

is a time update ID obtained from the JULIANTIMESTAMP procedure. It should be used with <mode> 2 and 3 to avoid conflicting changes.

Condition Code Settings

< (CCL) indicates insufficient capability.

= (CCE) indicates the SETSYSTEMCLOCK was successful.

> (CCG) indicates that there is a mismatch with <tuid>; retry after redetermining the relative error.

Example

```
CALL SETSYSTEMCLOCK ( JULIAN^GMT , MODE , TUID );
```

Related Programming Manual

None

SHIFTSTRING PROCEDURE

The SHIFTSTRING procedure upshifts or downshifts all alphabetic characters in a string. Nonalphabetic characters remain unchanged.

The syntax for SHIFTSTRING is:

```
CALL SHIFTSTRING ( <string>                ! i, o
                 ,<count>                  ! i
                 ,<casebit> );             ! i
```

<string> input, output

STRING:ref:*

is the character string to be shifted.

<count> input

INT:value

is the length of the string in bytes.

<casebit> input

INT:value

specifies a value indicating whether to upshift or downshift the string:

- .<15> = 0 the procedure upshifts the string indicated, making all alphabetic characters uppercase.
- .<15> = 1 the procedure downshifts the string indicated, making all alphabetic characters lowercase.

Condition Code Settings

The condition code has no meaning following a call to SHIFTSTRING.

Example

```
CALL SHIFTSTRING ( COMMAND , COMMAND^LEN , CASE^BIT );  ! upshift
```

Related Programming Manual

For programming information about the SHIFTSTRING utility procedure, refer to the GUARDIAN Operating System Programmer's Guide.

SIGNALPROCESSTIMEOUT PROCEDURE

The SIGNALPROCESSTIMEOUT procedure sets a timer based on process execution time. When the time expires, the calling process receives an indication in the form of a system message on $RECEIVE.

The syntax for SIGNALPROCESSTIMEOUT is:

```
CALL SIGNALPROCESSTIMEOUT ( <timeout-value>            ! i
                          ,[ <param1> ]                ! i
                          ,[ <param2> ]                ! i
                          ,[ <tag> ]  );               ! o
```

<time-out-value> input

INT(32):value

specifies the time period, in .01-second units, after which a timeout message should be queued on $RECEIVE. This value must be greater than 0D.

<param1> input

INT:value

identifies the timeout message read from $RECEIVE.

<param2> input

INT(32):value

identifies the timeout message read from $RECEIVE (same purpose as <param1>).

<tag> output

INT(32):ref:1

returns an identifier associated with the timer. This <tag> should not be used other than to call CANCELPROCESSTIMEOUT.

Condition Code Settings

< (CCL) indicates that SIGNALPROCESSTIMEOUT is unable to allocate a timer list element (TLE).

= (CCE) indicates that SIGNALPROCESSTIMEOUT is successful.

> (CCG) indicates that the given timeout value is illegal or that there is a bounds error on <tag>.

Considerations

- SIGNALPROCESSTIMEOUT and CANCELPROCESSTIMEOUT

  SIGNALPROCESSTIMEOUT can be used with CANCELPROCESSTIMEOUT using a multithreaded I/O process to verify that an I/O operation completes within a certain process execution time. The process calls SIGNALPROCESSTIMEOUT before initiating the I/O operation, then calls CANCELPROCESSTIMEOUT after completion if the process has not been signaled on $RECEIVE.

- The SIGNALPROCESSTIMEOUT procedure measures the time the process is executing. This procedure excludes the time spent by the CPU processing interrupts (and most micro-interrupts) while the process was running.

- Deadlock Possibility

  Consider the following:

```
CALL SIGNALPROCESSTIMEOUT (10000D,,,TLE);
CALL READ (REC^NUM, BUFFER, 4);
          .
          .  ! open number of $RECEIVE.
```

  The READ causes the process to stop and wait for the system message to be generated by timeout (assume no other messages are expected). Timeout does not occur unless the process executes for 100 seconds, causing a deadlock.

NOTE

This does not happen with SIGNALTIMEOUT, which operates on a real-time basis as opposed to process time.

Message

- Time Signal Message

  When a TLE set by a call to SIGNALPROCESSTIMEOUT times out, a system message (-26) is placed on the $RECEIVE queue to be read by the caller. (This message is identical to the message generated by SIGNALTIMEOUT except that the message number is different.)

Example

```
CALL SIGNALPROCESSTIMEOUT( VALUE , MSG, , TIMERTAG );
```

Related Programming Manual

None

SIGNALTIMEOUT PROCEDURE

The SIGNALTIMEOUT procedure sets a timer to a given number of units of elapsed time. When the time expires, the calling process receives an indication in the form of a system message on $RECEIVE.

The syntax for SIGNALTIMEOUT is:

```
CALL SIGNALTIMEOUT ( <timeout-value>                ! i
                   ,[ <param1> ]                    ! i
                   ,[ <param2> ]                    ! i
                   ,[ <tag> ] );                    ! o
```

<timeout-value> input

INT(32):value

specifies the time period, in .01-second units after which a timeout message should be queued on $RECEIVE. This value must be greater than 0.

<param1> input

INT:value

identifies the timeout message read from $RECEIVE.

<param2> input

INT(32):value

identifies the timeout message read from $RECEIVE (same purpose as <param1>).

<tag> output

INT:ref:1

returns an identifier associated with the timer. This <tag> should not be used other than to call CANCELTIMEOUT.

Condition Code Settings

< (CCL) indicates that SIGNALTIMEOUT is unable to allocate a time list element (TLE).

= (CCE) indicates that SIGNALTIMEOUT completed successfully.

> (CCG) indicates that the given timeout value is illegal.

Considerations

- SIGNALTIMEOUT and CANCELTIMEOUT

  SIGNALTIMEOUT can be used with CANCELTIMEOUT using a multithreaded I/O process to verify that an I/O operation completes within a certain time. The process calls SIGNALTIMEOUT before initiating the I/O operation, then calls CANCELTIMEOUT after completion if the process has not been signaled on $RECEIVE.

- The SIGNALTIMEOUT procedure measures actual elapsed time (the wall clock) that this process executes. This includes the time spent by the CPU in process code, system code, and processing interrupts that occurred while the process was running.

Message

- Time Signal Message

  When a timer times out, a system message (-22) is placed on the $RECEIVE queue to be read by the caller.

Example

```
CALL SIGNALTIMEOUT( 1000D , , , TIMERTAG );   ! 10 seconds.
```

Related Programming Manual

None

SORTERROR PROCEDURE

SORTERROR provides the message text corresponding to the last SORT/MERGE error code returned. An error condition can be received from any of the SORT/MERGE interface procedures.

The syntax for SORTERROR is:

```
{ <status> := } SORTERROR ( <ctlblock>              ! i
{ CALL        }             ,<buffer>  );           ! i


<status>                    returned value

  INT

  returns the number of characters in the error message.


<ctlblock>                  input

  INT:ref:200

  is a 200-word global integer storage array named in
  SORTMERGESTART.


<buffer>                    input

  INT:ref:*

  is an integer array of 32 words minimum for the error message
  text.
```

Condition Code Settings

The condition code has no meaning following a call to SORTERROR.

Example

```
TEXTLEN := SORTERROR ( SORTBLOCK , OUTBUF );
```

Related Programming Manual

For programming information about the SORTERROR procedure, refer to the SORT/MERGE Users Guide.

SORTERRORDETAIL PROCEDURE

SORTERRORDETAIL provides the index in the <in-filename> array (parameter to SORTMERGESTART) of the input file being accessed when the latest error occurred.

The syntax for SORTERRORDETAIL is:

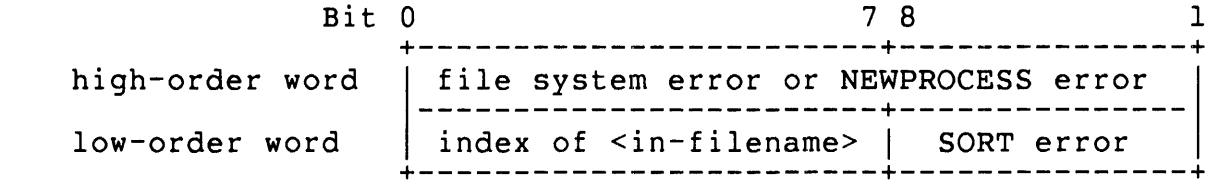
```
{ <status> := } SORTERRORDETAIL ( <ctlblock> );      ! i
{ CALL       }
```

<status> returned value

INT(32)

returns an INT(32) error code. The high-order word contains the file system error or a NEWPROCESS error. The low-order word contains the SORT error in the low-order byte and the index (in the <in-filename> parameter to SORTMERGESTART) of the input file that caused the error in the high-order byte.

If the error was not caused by an input file, or if the there is no outstanding error, the low-order word bits <0:7> contain 0. The format for the INT(32) is shown below:



(Refer to Appendix G for SORT/MERGE errors and the System Messages Manual for NEWPROCESS and file system errors.)

<ctlblock> input

INT:ref:200

is the global storage array named in SORTMERGESTART.

Condition Code Settings

The condition code has no meaning following a call to SORTERRORDETAIL.

Example

```
DETAIL^STATUS := SORTERRORDETAIL ( CONTROLBLOCK );
```

You only need to call this procedure if there is an error that the SORTERROR procedure recognizes.

Related Programming Manual

For programming information about the SORTERRORDETAIL procedure, refer to the SORT/MERGE Users Guide.

SORTMERGEFINISH PROCEDURE

SORTMERGEFINISH terminates the SORTPROG process after the sort completes. Any sorting or merging error stops the SORTPROG process at the time the error occurs. If SORTMERGEFINISH is called with "stop immediately" specified, the calling process receives a "process normal deletion (stop)" message in its $RECEIVE file when SORTPROG is deleted.

The syntax for SORTMERGEFINISH is:

```
{ <status> := } SORTMERGEFINISH ( <ctlblock>          ! i, o
{ CALL       }                  ,[ <abort> ]          ! i
                                ,[ <sparel> ]    ! error if used.
                                ,[ <spare2> ] ); ! error if used.
```

<status> returned value

INT

returns a SORT integer error code to indicate an error occurred; otherwise, a 0 returns (see Appendix G for SORT/MERGE errors).

<ctlblock> input

INT:ref:200

is the global storage array named in SORTMERGESTART.

<abort> input

INT:value

specifies when termination of SORTPROG occurs.

- 0 = specifies terminate SORTPROG after completion of the current sort. This is the default value.
- 1 = specifies stop SORTPROG immediately. The calling process receives a "process normal deletion (stop)" message in its $RECEIVE file.

→

<sparel> and <spare2>

If these parameters are used, an error is generated.

Condition Code Settings

The condition code has no meaning following a call to SORTMERGEFINISH.

Considerations

- If SORTPROG terminates due to an error, the user's program receives an error code when the next procedure is called.

Example

```
ERROR := SORTMERGEFINISH ( SORTBLOCK );
```

Related Programming Manual

For programming information about the SORTMERGEFINISH procedure, refer to the SORT/MERGE Users Guide.

SORTMERGERECEIVE PROCEDURE

SORTMERGERECEIVE returns the output records from the SORTPROG process directly to the user's program.

The syntax for SORTMERGERECEIVE is:

```
{ <status> := } SORTMERGERECEIVE ( <ctlblock>          ! i
{ CALL       }                   ,<buffer>           ! o
                                 ,<length>           ! o
                                 ,[ <spare1> ]     ! error if used.
                                 ,[ <spare2> ] ); ! error if used.
```

<status> returned value

INT

returns a SORT integer error code to indicate an error occurred; otherwise, a 0 returns (see Appendix G for a list of SORT/MERGE errors).

<ctlblock> input

INT:ref:200

is the global storage array named in SORTMERGESTART.

<buffer> output

INT:ref:*

is an integer array for the output records. The maximum record size from SORTMERGESTART determines the maximum length of this buffer.

<length> output

INT:ref:1

returns the length, in bytes, of the record retrieved. A -1, indicates all records returned.

→

<sparel> and <spare2>

If these parameters are used, an error is generated.

Condition Code Settings

The condition code has no meaning following a call to SORTMERGERECEIVE.

Considerations

- When the <out-filename> Parameter Is Omitted

  If the <out-filename> parameter is omitted or it consists of 24 spaces in the call to SORTMERGESTART, SORTMERGERECEIVE must be called to retrieve the records, one per call, into the user's program. The format of the returned record is determined by the <format> specified in the call to SORTMERGESTART. The formats SORTPROG provides are:

  --The entire record

  --The sequence number as a 32-bit (4-byte) integer

  --The key-field values strung together

  --The sequence number followed by the key field values

Example

```
CALL SORTMERGERECEIVE ( SORTBLOCK , RECBUF , LENGTH );
```

RECBUF receives the output record, and LENGTH is the number of bytes of the record retrieved.

Related Programming Manual

For programming information about the SORTMERGERECEIVE procedure, refer to the SORT/MERGE Users Guide.

SORTMERGESEND PROCEDURE

SORTMERGESEND provides input records from the user's program directly to the SORTPROG process.

The syntax for SORTMERGESEND is:

```
{ <status> := } SORTMERGESEND ( <ctlblock>              ! i
{ CALL        }                ,<buffer>                ! i
                               ,<length>                ! i
                               ,[ <stream-id> ]         ! i
                               ,[ <spare1> ]      ! error if used.
                               ,[ <spare2> ] );   ! error if used.
```

<status> returned value

INT

returns a SORT integer error code to indicate an error occurred; otherwise, a 0 returns (see Appendix G for a list of SORT/MERGE errors).

<ctlblock> input

INT:ref:200

is the global storage array named in SORTMERGESTART.

<buffer> input

INT:ref:*

is an integer array containing one input record.

<length> input

INT:value

is the length, in bytes, of the input record. This can vary among input records. It can be no smaller than the offset from the start of a record to the first character of the rightmost key, and it can be no larger than the longest input record length specified in the <in-filelen> parameter of SORTMERGESTART.

→

If sorting and <length> equals -1, there are no more input records.

If merging and <length> equals -1, there are no more records in the requested stream.

<stream-id> input

INT:ref:1

identifies the input stream where the next input record for SORTMERGESEND should come from. When all streams reply "no more records," a -1 returns to <stream-id> to indicate all input is entered and the process should continue to the next step. This parameter is required if any input file names for merging are left blank.

<spare-1> and <spare-2>

If these parameters are used, an error is generated.

Condition Code Settings

The condition code has no meaning following a call to SORTMERGESEND.

Considerations

- <in-filename> Parameter of SORTMERGESTART

  If the <in-filename> parameter is omitted or consists of all spaces in the call to SORTMERGESTART, the records for sorting must be provided through SORTMERGESEND, one call for each input record.

- Records cannot be supplied from both SORTMERGESEND and disc files for the same SORT run.

- Different SORT Runs for Sorting and Merging

  SORTMERGESEND can supply input records for sorting or for merging but cannot supply records for both operations in the same SORT run. If SORTMERGESEND is used to supply records to SORTPROG, SORTMERGESTART cannot specify input files for the same run.

Example

```
CALL SORTMERGESEND ( SORTBLOCK , INBUF , INLEN );
```

INBUF contains one input record, and INLEN is the number of bytes of the input record.

Related Programming Manual

For programming information about the SORTMERGESEND procedure, refer to the SORT/MERGE User's Guide.

SORTMERGESTART PROCEDURE

SORTMERGESTART initiates the SORTPROG process and passes parameter information. SORTMERGESTART is necessary in initiating all programmatic SORT or MERGE operations.

The syntax for SORTMERGESTART is:

```
{ <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
                                ,[ <spare1> ]       ! error if used
                                ,[ <spare2> ]       ! error if used
                                ,[ <spare3> ]       ! error if used
                                ,[ <spare4> ]       ! error if used
                                ,[ <spare5> ]  );   ! error if used
```

<status> returned value

INT

returns a SORT integer error code to indicate an error occurred; otherwise, a 0 returns (see Appendix G for a list of SORT/MERGE errors).

→

<ctlblock> input

INT:ref:200

is a 200-word global integer array supplied for storage of information about the SORT procedures. Values in <ctlblock> must not be altered between the call to SORTMERGESTART and the call to SORTMERGEFINISH; if they are, the user's program receives an error code.

<key-block> input

INT:ref:4

is an integer array defining the key fields. The first word is the total number of keys and the number of keys * 3 words. For a detailed discussion, see "Considerations," "<key-block> Key Fields."

<num-merge-files> input

INT:value

is the number of input files for merging. The maximum number of records SORTPROG accepts is limited by the amount of space available for the scratch file. The total number of files for both sorting and merging cannot exceed 32.

<num-sort-files> input

INT:value

is the number of input files for sorting. The maximum number of records SORTPROG accepts is limited by the amount of space available for the scratch file. The total number of files for both sorting and merging cannot exceed 32.

<in-file-name> input

INT:ref:*

is an array of 12-word entries, each naming a file of input records. An <in-file-name> is required for each input file. SORTPROG accepts a set of input files in the order presented, with the merge files first.

→

When working with more than one input file, SORTPROG uses the same key-field specifications for all input records.

If the source of the input records is not SORTMERGESEND, both sorted and unsorted records can be presented to SORTPROG.

If <in-file-name> is omitted or equals all spaces, SORTMERGESEND supplies the records from the user's program. See the SORTMERGESEND procedure.

If an <in-file-name> entry is for merging, it cannot be specified as the <out-file-name>.

<in-file-exclusion-mode> input

INT:ref:*

is an array of 16-bit entries. Each entry contains the exclusion mode used when SORTPROG opens the corresponding input file. If the exclusion mode for one <in-file-name> is specified, it must be provided for every <in-file-name>. The exclusion codes are:

```
-1 = use default
 0 = shared access
 1 = exclusive access
 3 = protected access
```

When an exclusion mode is not specified, the following defaults apply. If <in-file-name> is a:

```
nontemporary disc file  = protected
nondisc file            = exclusive
temporary disc file     = shared
terminal                = shared
```

<in-file-count> input

INT(32):ref:*

is an array of 32-bit entries. Each entry contains the maximum number of records in the corresponding <in-file-name>. If <in-file-count> is omitted or equals -1, SORTPROG uses the default number of records. If <in-filename> is:

→

- A structured disc file, the default is the number of records in <in-filename>.
- An unstructured, non-EDIT disc file, the default number of records is determined by dividing the end-of-file (EOF) length by the record length.
- An unstructured, EDIT disc file, the default number of records is determined by multiplying the EOF length by 2 and dividing by the record length.
- An unstructured disc file, the default record length is 132 bytes.

For nondisc files and records supplied by SORTMERGESEND, the default number of records is decimal 50,000.

<in-filelen> input

INT:ref:*

is an array of 16-bit entries. Each entry contains the largest record size in the corresponding <in-filename>. The largest record length allowed is 4080 bytes. If <in-filelen> is omitted or equals -1, SORTPROG uses the default record length.

This parameter can be omitted when <in-filename> is a structured disc file.

For unstructured files, nondisc files, and records supplied by SORTMERGESEND, the default for the largest record length is 132 bytes.

<format> input

INT:value

selects the output record format, where value:

0 = the output records are in the same format as the input. If <format> is omitted, this default is used.

1 = the output records are 32-bit integers describing the order of the sorted records. For example, if the twentieth input record is first in order after sorting, 20 is the value of the first output record.

→

2 = each output record is the characters in the key fields strung together in the order defined. Alphanumeric STRING key values are padded with blanks if they extend beyond the record length.

3 = each output record begins with the 32-bit (4-byte) record number concatenated with the key fields.

<out-filename> input

INT:ref:12

is a 12-word array that names the file for the output records. If <out-filename> is omitted or equals 24 spaces, SORTMERGERECEIVE returns the records, one at a time, to the user's program.

If <out-filename> is present and is a different file type than that requested, it is purged and a new file is created.

<out-file-exclusion-mode> input

INT:value

is the exclusion mode used when SORTPROG creates the output file. The exclusion mode codes are:

-1 = use default
 0 = shared access
 1 = exclusive access
 3 = protected access

When the exclusion mode is not specified, the following defaults apply if <out-filename> is a:

disc or magnetic tape file	= exclusive
temporary disc file	= shared
terminal	= shared

<out-file-type> input

INT:value

specifies the type of file SORTPROG creates for the output records. This parameter can be omitted if <out-filename> is omitted or equals 24 spaces, and SORTMERGERECEIVE is used to return the sorted records to the user's program. See the SORTMERGERECEIVE procedure.

The default for <out-file-type> is the same file type as the first input file. The <out-file-type> codes are:

-1 = null value (treat as if parameter is omitted)
0 = unstructured file
1 = relative file
2 = entry-sequenced file

SORTPROG does not send output to EDIT or key-sequenced files. For systems other than ENSCRIBE systems, the <out-file-type> is always unstructured.

<flags> input

INT:value

indicates to SORTPROG to perform a specific set of operations. The meanings of the flag fields are described in Table 2-6.

If SORTPROG exists and the current <process-start> parameters are changed, these changes are ignored, except for "priority."

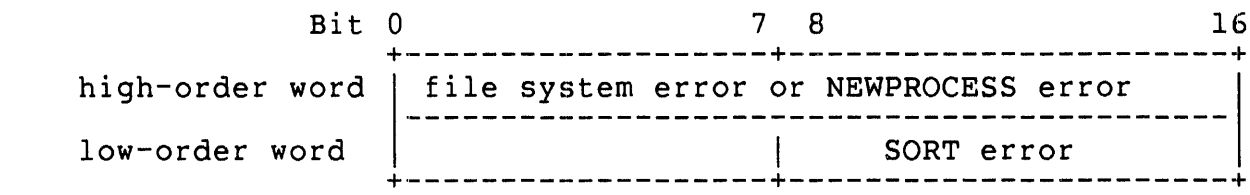
<errnum> output

INT(32):ref:1

returns a completion code of 0, if no error occurs, or an error status, if an error occurs. The high-order word contains the file system error number or a NEWPROCESS error number. The

→

low-order byte of the low-order word contains the SORT error code. The format for <errnum> is shown below:



Both the error code and the error procedure can be supplied. See the SORTERRORDETAIL procedure to obtain the index of the input file in the <in-filename> set. See the SORTERROR procedure to obtain the error message text corresponding to the error code. See Appendix G for SORT/MERGE errors and the System Messages Manual for NEWPROCESS and file system errors that are returned in <errnum>.

<errproc> input

PROCEDURE

is a procedure called by the SORT procedures when an error occurs. See "User Error Procedure," for a description of this procedure.

<scratch-filename> input

INT:ref:12

is a 12-word file name for the SORTPROG work file. The first 8 bytes can name the volume where SORTPROG creates the scratch file, and the remaining 16 bytes are left blank. If the scratch file exists or <scratch-filename> is supplied, it must be an unstructured file.

If this parameter is omitted or all blanks, SORTPROG creates a scratch file on volume $SYSTEM. See Table 2-6 (specifically, <flags>.<12:13>) for details on <scratch-filename> disposition.

→

<scratch-block> input

INT:value

is the length, in bytes, of the scratch file block. The length specified must be 512, 1024, 2048, or 4096 bytes. The scratch file block size must be large enough to accept the largest input record, rounded up to the nearest even byte, plus 14 bytes of overhead. The maximum block size allowed is 4096 bytes.

The default block size is the minimum of the largest block transfer allowed by the device containing the output and scratch files. The <scratch block> size cannot be larger than this default.

If <scratch-block> is omitted or equals a -1 value, the default block size is used.

<process-start> input

INT:ref:4

is a four-word array that supplies the information to start the SORTPROG process.

<word>[0] = priority, assigns the priority of the SORTPROG process. The default is the same priority as the calling process.

[1] = memory, specifies the maximum number of data pages allowed for the SORTPROG process. The default value is the larger of four pages or the amount of memory allocated to the user's program.

[2] = processor, selects the processor where SORTPROG runs. If omitted, SORTPROG runs in the same processor as the calling program.

[3] = system number, specifies in which system SORTPROG runs. The default is the current system (See the LOCATESYSTEM procedure).

The default is -1 = not passing parameters to NEWPROCESS.

→

<max-record-length> input

INT:ref:*

is the size, in bytes, of the largest output record for <out-filename> or for SORTMERGERECEIVE. The maximum record length returnable is 4080 bytes.

<collate-sequence-table> input

STRING:ref:256

is a 256-byte array defining the collating sequence used in the SORT run. This parameter applies to alphanumeric STRING items only. Each alphanumeric character is used as an index into the collating table to obtain the value used for comparison.

This parameter is ignored unless the "collating sequence table," flag <10>, is set to 1.

<sparel> and <spare5>

INT:value

If these parameters are used, an error is generated.

Condition Code Settings

The condition code has no meaning following a call to SORTMERGESTART.

Considerations

- <key-block> Key Fields

  The first word of the <key-block> array is the total number of keys, then the number of keys * 3 words. Every three words describes one key as follows:

```
            ------------------------------
word [0]   |     Number of keys           |
            ------------------------------
word [1]   |  Description of first key    |
word [2]   |      (three words)           |
word [3]   |                              |
            ------------------------------
word [4]   | Description of second key    |
word [5]   |      (three words)           |
word [6]   |                              |
            ------------------------------
    .                    .
    .                    .
    .                    .
```

  Each three-word key descriptor has the form:

```
                                    1 1 1 1 1 1
       Bit 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
          --+-+-+-+-+-+-+-+-----------------
Word [0] |O|U|R|R|R|R|R|R| Type            |
          ---------------------------------
Word [1] |             Length              |
          ---------------------------------
Word [2] |             Offset              |
          ---------------------------------
```

Word	Bits	Value	
[0]	<0>	O:	0 - ascending
			1 - descending
	<1>	U:	0 - do not upshift
			1 - upshift (Alphanumeric string only)
	<2:7>	R:	0 - reserved (must be 0)
	<8:15>	Type:	1 - ALPHANUMERIC STRING
			2 - UNSIGNED NUMERIC STRING
			3 - NUMERIC STRING SIGN TRAILING EMBEDDED
			4 - NUMERIC STRING SIGN TRAILING SEPARATE
			5 - NUMERIC STRING SIGN LEADING EMBEDDED
			6 - NUMERIC STRING SIGN LEADING SEPARATE
			9 - BINARY SIGNED
			10 - BINARY UNSIGNED
			11 - FLOAT

[1] <0:15> Length: key length in number of bytes. For key type 11, length must be 4 or 8 bytes. For key types 3 through 6, length must be 32 bytes or less.

[2] <0:15> Offset: number of bytes from beginning of record to key (record begins at 0). For key type 11, offset must be an even number.

- <flags> Fields

  Table 2-6 shows the flag bits for the <flags> fields.

Table 2-6. SORTMERGESTART <flags> Fields

Operation	⟨flags⟩ Bit	Value	Description
Scratch File Size Check	.⟨9⟩	0	The named scratch file is checked to see if it is large enough to contain the data. This is the default value.
		1	The named scratch file is not checked before it is used.
Collating Sequence Table	.⟨10⟩	0	The collating sequence parameter is ignored; compare normally. This is the default value.
		1	Use the alternate collating sequence table. If this flag is set and no alternate collating table is provided, an error occurs.
Remove Duplicates	.⟨11⟩	0	Keep records with keys that are duplicates. This is the default value.
		1	Do not keep records with keys that are duplicates. The first record is in the output; any following duplicate records are removed.
Save Scratch File	.⟨12⟩	0	Purge the scratch file after the SORT ends. This is the default value.
		1	Keep the scratch file after the SORT ends. Allowed only if the scratch file is named.
Scratch Must Be New	.⟨13⟩	0	If the scratch file exists, it is used after a PURGEDATA operation. This is true if the size of the scratch file required is equal to or smaller than the existing scratch file. If the existing scratch file is not used, it is purged. This is the default value.
		1	If the scratch file exists, purge it and create a new scratch file. This is not possible if the scratch file is a temporary file created by the user's program.
Out File Must Be New	.⟨14⟩	0	If ⟨out-file-name⟩ exists, use it after a PURGEDATA operation. The existing ⟨out-filename⟩ must agree in ⟨out-file-type⟩ and ⟨max-record-length⟩ and must be large enough to accept the sum of all the records from all the input files. This is the default value.
		1	If ⟨out-filename⟩ exists, purge it and create a new ⟨out-filename⟩. This is illegal for temporary files.
Restart	.⟨15⟩	0	Create a new SORTPROG process. This is the default value.
		1	Do not create a new SORTPROG process; use the existing one.
If SORTPROG terminates and the "restart" flag is set to 1, upon restarting, the current ⟨process-start⟩ parameters are used. If SORTPROG exists and the current ⟨process-start⟩ parameters are changed, these changes are ignored, except for "priority."			

- To reuse <out-filename> when it is a disc file that exists:

  --The current <out-filename> type must agree with the new <out-file-type>.

  --The current <out-filename> size must be equal to or greater than the sum of the sizes of all the input files.

  --The current <max-recordlen> must be equal to or greater than the maximum record length returned from SORTMERGESTART.

  --The "<out-filename> must be new" flag (<flag>.<14>) must be set to 0.

- Caution Using <ctlblock>

  Data in the control block can change without warning. The user must not use the information stored here.

- Input File

  --SORTPROG accepts all file types except processes.

  --The maximum number of input files SORTPROG accepts is 32. The files may contain fixed- or variable-length records.

  --The sum of the <number-merge-files> and <number-sort-files> parameters must be at least 1. Therefore, one of the two parameters must be specified, but neither one is specifically required.

  --SORTPROG does not work on blocked tape files. Before presenting these files to SORTPROG, use the File Utility Program (FUP) to deblock the records. Refer to the GUARDIAN Operating System Utilities Reference Manual for information about FUP.

- Output File Types (<out-filename>)

  If the output file is a disc file, SORTPROG creates it according to the following:

  1. SORTPROG uses the file type specified in the <out-file-type> parameter.

  2. SORTPROG uses the existing <out-filename> file type, unless it is an EDIT or key-sequenced file.

  3. SORTPROG uses the first <in-filename> file type, unless it is an EDIT, key-sequenced, or nondisc file.

  4. If none of the above conditions exist, an entry-sequenced file is created.

For systems other ENSCRIBE systems, the default output file type is always unstructured.

SORTPROG does not send output to EDIT or key-sequenced files.

If <out-filename> is a magnetic tape, records are written one per block.

- Record Count

  The <in-file-count> need not be the exact number of records in the file, but when rounding off the value, round up and not down.

- Memory Size

  Choosing the correct amount of memory is important for good performance; which also, it depends on the quantity of data and the current activity in the selected processor. SORTPROG performance continues to improve with increasing memory allocation, until no more physical memory is available and page faulting increases sharply. The memory allocation for SORTPROG can be decreased if SORT operations tend to degrade the performance of other processes.

  When allocating memory for SORT operations, the programmer should consider the following: the size of the scratch file block, the input record size, the amount of physical memory available, and the priorities of other processes relative to SORTPROG.

- Restart Option

  This option allows successive SORT runs without creating a new SORTPROG process.

  A call to SORTMERGESTART returns immediately after SORTPROG reads the input parameters. Before calling SORTMERGESTART to restart, SORTMERGESTATISTICS must be called or SORTPROG must have terminated with an error message. SORTPROG accepts parameters for restart in the following way:

  --If SORTPROG terminates abnormally and the "restart" flag is set to 1 ("do not create a new SORTPROG process, use the existing one"), upon restarting, the most current parameters specified are used.

  --If SORTPROG exists and the current <process-start> parameters are changed, these changes are ignored, except for "priority."

- Alternate Collating Sequence Table

  An alternate collating sequence table can be obtained by reading the file produced by the COLLATEOUT command in the conversational mode SORT program.

User Error Procedure

This is a user-written procedure that the SORT/MERGE interface procedures can call when an error condition is detected by SORTPROG.

The syntax for this error procedure is:

```
PROC  <errproc> ( <code> );
```

<errproc>

is the procedure named in the <errproc> parameter parameter of SORTMERGESTART.

<code>

INT(32):value

describes the error and is identical to the value in the <errnum> parameter of SORTMERGESTART. Refer to the SORT/MERGE User's Guide for more information about SORT errors and messages.

The following is an example of the user error procedure:

```
PROC sorterrproc (errcode);
  INT(32) errcode;
  BEGIN
          .       ! Tandem Application Language
          .       ! statements of your error routine.
          .
  END;
          .
          .
          .
  CALL SORTMERGESTART ( SORTBLOCK , KEYS
                      ,             ! number of files to
                                    ! merge.
                      ,NUMSORT
                      ,INFILE
                      ,             ! in file exclusion mode.
                      ,             ! in file count.
                      ,             ! in file length.
                      ,             ! format.
                      ,OUTFILE
                      ,             ! out file exclusion mode.
                      ,             ! out file type.
                      ,             ! flags for operation.
                      ,             ! error number.
                      , SORTERRPROC );
          .
          .
          .
```

Example

See "Considerations."

Related Programming Manual

For programming information about the SORTMERGESTART procedure, refer to the SORT/MERGE Users Guide.

SORTMERGESTATISTICS PROCEDURE

SORTMERGESTATISTICS is called at the conclusion of a successful SORT run to obtain information about that run.

The syntax for SORTMERGESTATISTICS is:

```
{ <status> := }  SORTMERGESTATISTICS ( <ctlblock>          ! i
{ CALL        }                      ,<length>            ! i, o
                                     ,<statistics>        ! o
                                     ,[ <spare1> ]     ! do not use
                                     ,[ <spare2> ] ); ! do not use
```

<status> returned value

INT

returns a SORT integer error code to indicate an error occurred; otherwise, a 0 returns (see Appendix G for a list of SORT/MERGE errors).

<ctlblock> input

INT:ref:200

is the same global storage array named in SORTMERGESTART.

<length> input, output

INT:ref:*

indicates the length, in words, of the statistics returned from the SORTPROG process after run completion. When statistics return, <length> is set to the number of words actually returned. The default is a 0 value for <length>, which means no statistics returns.

<statistics> output

INT:ref:21

is a 21-word array into which the SORTPROG statistics return (see "Considerations" for a description of this array).

Condition Code Settings

The condition code has no meaning following a call to SORTMERGESTATISTICS (see the <status> parameter).

Considerations

- 21-Word Array--<statistics>

```
INT      <word>[0] = RECORD SIZE, maximum record size in bytes
               [1] = BUFFER PAGES, number of 1024-word pages of
                     memory actually used by SORTPROG as a SORT
                     area

INT(32)      [2:3] = RECORDS, number of records
             [4:5] = ELAPSED TIME, in .01-seconds, time spent
                     sorting
             [6:7] = COMPARES, number of times two records
                     were compared
             [8:9] = SCRATCH SEEKS, number of READs and WRITEs
                     on the scratch file
           [10:11] = I/O WAIT TIME, time spent, in .01-seconds,
                     in calls to READ, WRITE, and AWAITIO in SORTPROG
           [12:13] = SCRATCH DISC, number of bytes in the
                     scratch file
           [14:15] = INITIAL RUNS, number of runs generated
                     by the first pass

INT           [16] = FIRST MERGE ORDER, number of runs merged in
                     the first intermediate pass
              [17] = MERGE ORDER, maximum number of runs that
                     can be merged
              [18] = INTERMEDIATE PASSES, number of MERGE cycles
                     between initial run formation and final MERGE

INT(32)    [19:20] = NUMBER OF DUPLICATES, number of duplicate
                     records removed.
```

Example

```
CALL SORTMERGESTATISTICS ( SORTBLOCK , LENGTH , STATISTICS );
```

Related Programming Manual

For programming information about the SORTMERGESTATISTICS procedure, refer to the SORT/MERGE Users Guide.

SPOOLCONTROL PROCEDURE

The SPOOLCONTROL procedure is used to perform device-dependent I/O operations when the application process is spooling at level 3.

If a level 3 buffer is specified in a call to SPOOLSTART, the SPOOLCONTROL procedure must be used in place of the CONTROL procedure.

The syntax for SPOOLCONTROL is:

```
<error-code> := SPOOLCONTROL ( <level-3-buff>                  ! i, o
                             ,<operation>                      ! i
                             ,<param>                          ! i
                             ,[ <bytes-written-to-buff> ] );!  o
```

<error-code> returned value

INT

returns one of the following spooler error codes:

0	= successful operation
%1000-%1377	= error on file to collector (<8:15> contains a file system error code number; see "Considerations")
10000	= missing parameter
10001	= parameter is present, but its content is wrong
11000	= checkpoint exit
11001	= attempted to write to the collector without first opening the file

<level-3-buff> input, output

INT:ref:512

is the <level-3-buffer> specified in the SPOOLSTART procedure.

<operation> input

INT:value

is a CONTROL operation value (see the CONTROL procedure and Appendix A about control operations).

→

<param> input

INT:value

is a <parameter> for the specified CONTROL operation (see Appendix A).

<bytes-written-to-buff> output

INT:ref:1

returns the number of bytes to be checkpointed from the <level-3-buff>. This is a NonStop consideration.

Condition Code Settings

The condition code has no meaning following a call to SPOOLCONTROL (see the <error-code> parameter).

Considerations

- When SPOOLCONTROL Should Be Recalled

  If flags.<11> of SPOOLSTART is set to 1, a return of %11000 from SPOOLCONTROL indicates that the <level-3-buff> is about to be written to the collector. The buffer should be checkpointed, and SPOOLCONTROL should be recalled.

- SPOOLCONTROL--Bits <8:15>--Error on File to Collector

  Some file system errors have special significance to a process sending data to a collector; these errors can be found in the System Messages Manual.

  A program using level 1 or level 2 spooling gets these errors from the WRITE or OPEN procedures, while a program spooling at level 3 obtains these errors in bits <8:15> of a spooler error code in the %1000 range.

Example

```
ERROR := SPOOLCONTROL ( COLL^BUFF , 5 );
```

Related Programming Manual

For programming information about the SPOOLCONTROL procedure, refer to the Spooler Programmer's Guide.

SPOOLCONTROLBUF PROCEDURE

The SPOOLCONTROLBUF procedure is used to perform device-dependent I/O operations requiring a data buffer when the application process is spooling at level 3.

This procedure must be used in place of CONTROLBUF if a level 3 buffer is specified in a call to SPOOLSTART.

The syntax for SPOOLCONTROLBUF is:

```
<error-code> := SPOOLCONTROLBUF ( <level-3-buff>              ! i, o
                                 ,<operation>                 ! i
                                 ,<buffer>                    ! i
                                 ,<count>                     ! i
                                ,[ <bytes-written-to-buff> ] ); ! o
```

<error-code> returned value

INT

returns one of the following spooler error codes:

0	= successful operation
%1000-%1377	= error on file to collector (<8:15> contains a file system error code number; see "Considerations")
%10000	= missing parameter
%10001	= parameter is present, but its content is wrong
%11000	= checkpoint exit
%11001	= attempted to write to the collector without first opening the file

<level-3-buff> input, output

INT:ref:512

is the <level-3-buff> specified in the SPOOLSTART procedure.

<operation> input

INT:value

is a CONTROLBUF operation (see Table 2-7 under "Considerations").

→

<buffer> input

INT:ref:*

is an array containing the control information to be sent to the print device.

<count> input

INT:value

is the number of bytes of information contained in the buffer.

<bytes-written-to-buff> output

INT:ref:1

returns the number of bytes to be checkpointed from the <level-3-buff>. This is a NonStop consideration.

Condition Code Settings

The condition code has no meaning following a call to SPOOLCONTROLBUF (see the <error-code> parameter).

Considerations

- Table 2-7 shows the value of SPOOLCONTROLBUF operations value.

Table 2-7. SPOOLCONTROLBUF Operations

<operation>	Definition
1	load DAVFU (printer subtype 4) <buffer> = VFU buffer to be loaded <count> = number of bytes contained in <buffer>

- When SPOOLCONTROLBUF Should Be Recalled

  If flags.<11> of SPOOLSTART is set to 1, a return of %11000 from SPOOLCONTROLBUF indicates that the <level-3-buff> is about to be written to the collector. The buffer should be checkpointed, and SPOOLCONTROLBUF should be recalled.

- SPOOLCONTROLBUF--Bits <8:15>--Error on File to Collector

  Some file system errors have special significance to a process sending data to a collector; these errors can be found in the System Messages Manual.

  A program using level 1 or level 2 spooling gets these errors from the WRITE or OPEN procedures, while a program spooling at level 3 obtains these errors in bits <8:15> of a spooler error code in the %1000 range.

Example

```
ERROR := SPOOLCONTROLBUF ( COLL^BUFF , 1 , CON^BUFF , COUNT );
```

Related Programming Manual

For programming information about the SPOOLCONTROLBUF procedure, refer to the Spooler Programmer's Guide.

SPOOLEND PROCEDURE

The SPOOLEND procedure can be used to complete a job being spooled at level 3.

The SPOOLEND procedure writes any remaining data in the collection process buffer to the collector, sends the collection process a termination message, and optionally modifies the job's attributes.

The syntax for SPOOLEND is:

```
<error-code> := SPOOLEND ( <level-3-buff>              ! i
                         ,[ <flags> ] );               ! i


<error-code>                 returned value

  INT

  returns one of the following spooler error codes:

              0 = successful operation
    %1000-%1377 = error on file to collector (<8:15> contains
                  a file system error code number; see
                  "Considerations")
         %10000 = missing parameter
         %10001 = parameter is present, but its content is wrong
         %11000 = checkpoint exit
         %11001 = attempted to write to the collector without
                  first opening the file


<level-3-buff>               input

  INT:ref:512

  is the <level-3-buffer> specified in the call to SPOOLSTART.
```

→

```
<flags>                          input

  INT:value

  overrides the flags specified in SPOOLSTART.  If bit <8> is
  set, the job is canceled rather than printed.

    flags.<0:7>    = reserved for use by the collector
         .<8>      = cancel job flag:  0 = off
                                       1 = on
         .<9>      = HOLD flag:  0 = off
                                 1 = on
         .<10>     = HOLDAFTER flag:  0 = off
                                      1 = on
         .<11:12>  = reserved; must be 0
         .<13:15>  = job priority

  If this parameter is omitted, the attributes established by the
  call to SPOOLSTART are not changed.
```

Condition Code Settings

The condition code has no meaning following a call to SPOOLEND (see the <error-code> parameter).

Considerations

- A call to SPOOLEND causes any existing data in the collection process buffer to be written to the collector.
- Following a call to SPOOLEND, a new job can be started without reopening a file to the collector.
- SPOOLEND must be called to terminate the spooling of any job being spooled at level 3 (using the spooler interface procedures).
- Calling SPOOLEND with the <flags> cancel bit set to 1 has the same effect as never having started the job.
- Including <flags> causes all bit fields to override the values specified in the <flags> parameter of SPOOLSTART.

- SPOOLEND--Bits <8:15>--Error on File to Collector

  Some file system errors have special significance to a process sending data to a collector; these errors can be found in the System Messages Manual.

  A program using level 1 or level 2 spooling gets these errors from the WRITE or OPEN procedures, while a program spooling at level 3 obtains these errors in bits <8:15> of a spooler error code in the %1000 range.

Example

```
SPERRNUM := SPOOLEND ( BUFFER , %B0000000001000100 );
   ! put job on HOLD and set priority 4.
```

Related Programming Manual

For programming information about the SPOOLEND procedure, refer to the Spooler Programmer's Guide.

SPOOLERCOMMAND PROCEDURE

The SPOOLERCOMMAND procedure is used to perform SPOOLCOM and PERUSE operations from within programs.

The SPOOLERCOMMAND procedure allows a process to send a SPOOLCOM command to the spooler supervisor.

The syntax for SPOOLERCOMMAND is:

```
<error-code> := SPOOLERCOMMAND ( <filenum-to-supervisor>      ! i
                               ,<command-code>                ! i
                               ,[ <command-param> ]           ! i
                               ,<subcommand-code>             ! i
                               ,[ <subcommand-param> ] );     ! i
```

<error-code> returned value

INT

returns one of the following spooler error codes:

0	= successful operation
%3000-%3377	= error on file to supervisor (file system error in bits <8:15>), refer to the System Messages Manual.
%10000	= parameter missing
%10001	= parameter in error
%14000	= invalid command
%14001	= command parameter missing
%14002	= command parameter in error
%14003	= invalid subcommand
%14004	= subcommand missing
%14005	= subcommand parameter in error
%14010	= cannot add entry to tables
%14011	= cannot find entry requested
%14012	= entry not in proper state for requested operation
%14013	= entry in use; cannot be deleted
%14014	= security violation
%14015	= process not a spooler supervisor

→

<filenum-to-supervisor> input

INT:value

is the number of an open supervisor file used by the caller. The file number is returned from a previous call to OPEN.

<command-code> input

INT:value

is the number of the command to be executed (see Table 2-8).

<command-param> input

INT:ref:*

is a buffer containing the parameter to the command being executed. The size and content for each command are shown in Table 2-6.

<subcommand-code> input

INT:value

is the number of the subcommand to be executed (the <subcommand-codes> are listed in Table 2-8).

<subcommand-param> input

INT:ref:*

is a buffer containing the parameter to the subcommand being executed. Table 2-8 shows the size and content for each subcommand.

Table 2-8 shows the SPOOLERCOMMAND parameters: <command-code>, <command-param>,<subcommand-code>, and <subcommand-param>.

Table 2-8. SPOOLERCOMMAND--Command and Subcommand Parameters (Continued)

SPOOLCOM Command	SPOOLCOM Subcommand	SPOOLERCOMMAND Parameters ⟨cmd-code⟩	⟨cmd-param⟩	⟨subcmd-code⟩	⟨subcmd-param⟩
DEV	blank	1	⟨device-name⟩ INT:16	100	none
	SPEED			101	⟨lpm⟩ (INT: 1)
	PROCESS			102	⟨process-name⟩ (STRING: 8)
	EXCLUSIVE			103	0 = off 1 = on 2 = off } (INT: 1)
	FIFO			104	0 = off 1 = on } (INT: 1)
	RETRY			105	⟨interval⟩ (INT:1)
	TIMEOUT			106	⟨num retries⟩ (INT:1)
	FORM			107	[⟨form name⟩] (STRING:8)
	SKIP			108	+ ⟨num-pages⟩ − ⟨num-pages⟩ } (INT: 1)
	SKIPTO			109	⟨page-num⟩ (INT:1)
	HEADER			110	0 = off 1 = on 2 = batch } (INT: 1)
	SUSPEND			111	none
	CLEAR			112	1 = DEL 0 = no DEL } (INT: 1)
	ALIGN			113	none
	DRAIN			114	none
	START			115	none
	DELETE			116	none
	JOB			117	⟨job-code⟩ (INT:1)
	TRUNC			118	0 = off 1 = on } (INT: 1)
	WIDTH			119	⟨device-width⟩ (INT:1)
	RESTART			120	none
	PARM			148	⟨device-param⟩ (INT:1)
JOB	FORM	2	job number INT:1-word	107	[⟨form-name⟩] (STRING:8)
	START			115	none

Table 2-8. SPOOLERCOMMAND--Command and Subcommand Parameters

SPOOLCOM Command	SPOOLCOM Subcommand	SPOOLERCOMMAND Parameters ⟨cmd-code⟩	⟨cmd-param⟩	⟨subcmd-code⟩	⟨subcmd-param⟩
	DELETE			116	none
	HOLDAFTER			121	0 = off 1 = on } (INT: 1)
	HOLD			122	none
	COPIES			123	⟨num-copies⟩ (INT:1)
	REPORT			124	[⟨report-name⟩] (STRING:8)
	LOC			125	[⟨location-name⟩] (INT:8)
	SELPRI			126	⟨selection⟩ ⟨priority⟩ } (INT: 1)
	OWNER			127	⟨group⟩ = ⟨0:7⟩ ⟨user⟩ = ⟨8:15⟩ } (INT: 1)
LOC	blank	3	⟨group.dest⟩ INT:16	100	none
	DELETE		⟨group⟩.[⟨dest⟩] INT:16	116	none
	DEV		[⟨group⟩].⟨dest⟩ INT:16	131	[⟨device-name⟩] (INT:16)
	BROADCAST		⟨group⟩ INT:16	132	0 = off 1 = on } (INT: 1)
COLLECT	blank	4	⟨collection-process⟩ INT:3	100	none
	DRAIN			114	none
	START			115	none
	DELETE			116	none
	FILE			141	⟨program-filename⟩ (STRING:12)
	CPU			142	⟨cpu⟩ (INT:1)
	BACKUP			143	⟨backup-cpu⟩ (INT:1)
	PRI			144	⟨process-priority⟩ (INT:1)
	DATA			145	⟨data-filename⟩ (STRING:12)
	UNIT			146	⟨unit-size⟩ (INT:1)
PRINT	blank	5	⟨print-process⟩ INT:3	100	none
	START			115	none
	DELETE			116	none

Table 2-8. SPOOLERCOMMAND--Command and Subcommand Parameters (Continued)

SPOOLCOM Command	SPOOLCOM Subcommand	SPOOLERCOMMAND Parameters ⟨cmd-code⟩	⟨cmd-param⟩	⟨subcmd-code⟩	⟨subcmd-param⟩
	FILE			141	⟨program-filename⟩ (STRING:12)
	CPU			142	⟨cpu⟩ (INT:1)
	BACKUP			143	⟨backup-cpu⟩ (INT:1)
	PRI			144	⟨process-priority⟩ (INT:1)
	DEBUG			147	0 = off, 1 = on (INT: 1)
	PARM			148	⟨print-process-param⟩ (INT:1)
SPOOLER	DRAIN	6	none	114	none
	START			115	none
	ERRLOG			151	⟨filename⟩ (STRING:12)

- Device name reflects the internal format of the device file. There are two forms of the device name, depending on whether you are dealing with a logical device or a process.

  Logical device:

  device[0:3] = \⟨sysname⟩ blank-filled
  [4:7] = $⟨devname⟩ blank-filled
  [8:11] = #⟨subdev⟩ blank-filled
  [12:15] = blank-filled.

  Process:

  name[0:3] = \⟨sysname⟩ blank-filled
  [4:7] = $⟨process-name⟩ blank-filled
  [8:11] = #⟨1st-qualifier⟩ blank-filled
  [12:15] = ⟨2nd-qualifier⟩ blank-filled.

- The blanks parameter can be used to create a device, a location, a collector, or a print process with all default attributes.
- Parameters requiring an on or off parameter use a word containing 0 for off and 1 for on.
- Location name parameter is in internal format:

  location[0:3] = #⟨group-name⟩ blank-filled
  [4:7] = ⟨destination-name⟩ blank-filled.

  If you want to leave out either the group name or destination name, just fill that field with blanks.

- Filename is in the internal format:

  filename[0:3] = $⟨disc-name⟩ blank-filled
  [4:7] = ⟨volname⟩ blank-filled
  [8:11] = ⟨filename⟩ blank-filled

- For a detailed description of the command and subcommand codes that are sent to the supervisor through the SPOOLERCOMMAND procedure, refer to the *GUARDIAN Operating System Utilities Reference Manual.*

Condition Code Settings

The condition code has no meaning following a call to SPOOLERCOMMAND (see the <error-code> parameter).

Considerations

- Note that <subcommand-code> is a required parameter on every call to SPOOLERCOMMAND.
- Commands With No Subcommands

  Commands not accompanied by subcommands should have code 100 as their <subcommand-code> (for example, creating a component with all default parameters).

Example

```
COM^ERROR := SPOOLERCOMMAND ( FILENUM
                            , COM^CODE
                            ,                ! command parameter.
                            , SUB^CODE );
```

Related Programming Manual

For more information about the SPOOLERCOMMAND utility procedure, refer to the Spooler Programmer's Guide.

SPOOLEREQUEST PROCEDURE

The SPOOLEREQUEST procedure allows a perusal process to access a spooled job outside the control of the spooler supervisor.

The syntax for SPOOLEREQUEST is:

```
<error-code> := SPOOLEREQUEST ( <supervisor-filenum>          ! i
                              ,<job-num>                      ! i
                              ,<print-control-buffer> );      ! o
```

<error-code> returned value

INT

returns one of the following spooler error codes:

0	= successful operation
%3000-%3377	= error on file to supervisor (file system error in bits <8:15>, refer to the System Messages Manual)
%10000	= parameter missing
%10001	= parameter in error
%14001	= command parameter missing
%14002	= command parameter in error
%14011	= cannot find entry requested
%14014	= security violation
%14015	= process not a spooler supervisor

<supervisor-filenum> input

INT:value

is the number of an open Supervisor file used by the caller. The file number is returned by a previous OPEN.

<job-num> input

INT:value

is the number of the job to be accessed.

→

<print-control-buffer> output INT:ref:64 returns a "start job" message suitable for passing to the PRINTREADCOMMAND procedure.

Condition Code Settings

The condition code has no meaning following a call to SPOOLEREQUEST (see <error-code> parameter).

Considerations

- The PRINTCOMPLETE, PRINTINFO, PRINTINIT, and PRINTSTATUS spooler print procedures cannot be used by a perusal process.
- SPOOLEREQUEST must be used in conjunction with the spooler print procedures (PRINTREADCOMMAND, PRINTREAD, and PRINTSTART) described in this manual.
- Attempting to Read Data That Does not Exist

  Since the supervisor does not know that the data file is being accessed, it allows the job to be deleted. If this occurs, PRINTREAD returns an "invalid data file" error (%12002) when attempting to read a line of data that is no longer there.
- Accessor ID and Caller of SPOOLEREQUEST

  SPOOLEREQUEST only returns job information if the accessor ID of the process calling SPOOLEREQUEST (for example, the user executing SPOOLEREQUEST) matches the job's owner.

Example

```
REQUEST^ERROR := SPOOLEREQUEST ( FILENUM , JOB , PRINT^BUFFER );
```

Related Programming Manual

For more information about the SPOOLEREQUEST utility procedure, refer to the Spooler Programmer's Guide.

SPOOLERSTATUS PROCEDURE

The SPOOLERSTATUS procedure is used to perform SPOOLCOM and PERUSE operations from within programs.

The SPOOLERSTATUS procedure allows a process to obtain the status of spooler components.

The syntax for SPOOLERSTATUS is:

```
<error-code> := SPOOLERSTATUS ( <supervisor-filenum>      ! i
                              ,<command-code>             ! i
                              ,<scan-type>                ! i
                              ,<status-buffer> );         ! i, o
```

<error-code> returned value

INT

returns one of the following spooler error codes:

0	= successful operation
%3000-%3377	= error on file to supervisor (file system error in bits <8:15>; refer to the System Messages Manual)
%10000	= parameter missing
%10001	= parameter in error
%14006	= end of SPOOLERSTATUS entries
%14007	= entry not found by SPOOLERSTATUS
%14015	= process not a spooler supervisor
%14016	= SPOOLERSTATUS request in progress

<supervisor-filenum> input

INT:value

is the number of an open supervisor file used by the caller. The file number is returned by a previous call to OPEN.

→

<command-code> input

INT:value

specifies the spooler component whose status is being sought. The range of values and their meanings are:

1 : device
2 : job
3 : location
4 : collector
5 : print process
6 : spooler
7 : jobs on a particular device queue
8 : occurrences of a particular job
9 : jobs with a particular location
10 : cross-reference by location
11 : cross-reference by device
12 : cross-reference by print process

<scan-type> input

INT:value

specifies the type of scan desired:

0 = status of the item specified in the <status-buffer>
1 = status of the item that follows the item specified in the <status-buffer>

<status-buffer> input, output

INT:ref:64

is a 64-word buffer where the status returns. The format of the status buffer depends on the particular command code (refer to the Spooler Programmer's Guide).

Condition Code Settings

The condition code has no meaning following a call to SPOOLERSTATUS (see the <error-code> parameter).

Considerations

- The combination of <command-code>, <scan-type>, and data in the <status-buffer> determines the status information returned. Refer to the Spooler Programmer's Guide for more information about status.

- If %14016 returns, call SPOOLERSTATUS again.

- Locations and New Groups

  If you request the status of a set of locations in the spooler system, and a new group is encountered (that is, the group number changes), you need to call SPOOLERSTATUS twice:

  1. The first call to SPOOLERSTATUS returns the group number with the destination blank.

  2. The second call to SPOOLERSTATUS returns the destination information about the group's location.

Example

```
STATUS^ERROR := SPOOLERSTATUS ( FILENUM , COM^CODE , TYPE , BUFF );
```

Related Programming Manual

For more information about the SPOOLERSTATUS utility procedure, refer to the Spooler Programmer's Guide.

SPOOLJOBNUM PROCEDURE

The SPOOLJOBNUM procedure returns the job number of the job currently being spooled to the collector. This procedure can be used when spooling at levels 1, 2, or 3.

The syntax for SPOOLJOBNUM is:

```
<error-code> := SPOOLJOBNUM ( <filenum-to-collector>        ! i
                            ,<job-num> );                   ! o


<error-code>                 returned value

  INT

  returns one of the following spooler error codes:

             0  = successful operation
    %1000-%1377 = error on file to collector (<8:15> contains
                  a file system error code number; see
                  "Considerations")
         %10000 = missing parameter
         %11001 = attempted to write to the collector without
                  opening the file first


<filenum-to-collector>     input

  INT:value

  is the file number of the collector obtained through a call
  to the GUARDIAN file system OPEN procedure.


<job-num>                  output

  INT:ref:1

  is the job number of the job currently being spooled to the
  collector through the specified file number.
```

Condition Code Settings

The condition code has no meaning following a call to SPOOLJOBNUM (see the <error-code> parameter).

Considerations

- A call to SPOOLJOBNUM can be issued by an application spooling at any level.
- When spooling at level 1, a job is not created until after a WRITE, SETMODE, or CONTROL procedure is called once.
- When spooling at levels 2 and 3, a job is not created until after the SPOOLSTART procedure is called.
- SPOOLJOBNUM--Bits <8:15>--Error on File to Collector

  Some file system errors have special significance to a process sending data to a collector; these errors are listed in the System Messages Manual.

  A program using level 1 or level 2 spooling gets these errors from the WRITE or OPEN procedures, while a program spooling at level 3 obtains these errors in bits <8:15> of a spooler error code in the %1000 range.

Example

```
ERROR := SPOOLJOBNUM ( FILENUM^COLL , JOB^NUM );
```

Related Programming Manual

For programming information about the SPOOLJOBNUM procedure, refer to the Spooler Programmer's Guide.

SPOOLSETMODE PROCEDURE

The SPOOLSETMODE procedure is used to set device-dependent functions when an application process is using the spooler interface procedures.

This procedure must be used in place of the SETMODE procedure if a level 3 buffer is specified in a call to SPOOLSTART.

The syntax for SPOOLSETMODE is:

```
<error-code> := SPOOLSETMODE ( <level-3-buff>                ! i, o
                             ,<function>                     ! i
                             ,[ <param1> ]                   ! i
                             ,[ <param2> ]                   ! i
                             ,[ <bytes-written-to-buff> ] ); ! o
```

<error-code> returned value

INT

returns one of the following spooler error codes:

0	= successful operation
%1000-%1377	= error on file to collector (<8:15> contains a file system error code number; see "Considerations")
%11000	= checkpoint exit
%11001	= attempted to write to the collector without first opening the file

<level-3-buff> input, output

INT:ref:512

is the <level-3-buff> specified in the call to SPOOLSTART.

<function> input

INT:value

is a SETMODE <function>. Setmode functions are listed in Appendix C.

⟶

<param1> and <param2> input

INT:value

are the parameters for the specified SETMODE function (see Appendix C).

<bytes-written-to-buff> output

INT:ref:1

returns the number of bytes to be checkpointed from the <level-3-buff>. This is a NonStop consideration.

Condition Code Settings

The condition code has no meaning following a call to SPOOLSETMODE (see the <error-code> parameter).

Considerations

- When SPOOLSETMODE Should Be Recalled

  If flags.<11> of SPOOLSTART is set to 1, a return of %11000 from SPOOLSETMODE indicates that the <level-3-buff> is about to be written to the collector. The buffer should be checkpointed, and SPOOLSETMODE should be recalled.

- SPOOLSETMODE--Bits <8:15>--Error on File to Collector

  Some file system errors have special significance to a process sending data to a collector; these errors are listed in the System Messages Manual.

  A program using level 1 or level 2 spooling gets these errors from the WRITE or OPEN procedures, while a program spooling at level 3 obtains these errors in bits <8:15> of a spooler error code in the %1000 range.

Example

```
ERROR := SPOOLSETMODE ( COLL^BUFF , 68 , 2 );  ! select expanded
                                               ! print.
```

Related Programming Manual

For programming information about the SPOOLSTART procedure, refer to the Spooler Programmer's Guide.

SPOOLSTART PROCEDURE

The SPOOLSTART procedure formats a spooler buffer suitable for passing to other spooler interface procedures. It can specify job attributes, or establish level 3 spooling, or both. This procedure establishes a level 2 or level 3 spooling session.

The syntax for SPOOLSTART is:

```
<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> returned value

INT

returns one of the following spooler error codes:

0	= successful operation
%1000-%1377	= error on file to collector (<8:15> contains a file system error code number; see "Considerations")
%10001	= parameter is present, but its content is wrong
%11000	= checkpoint exit
%11001	= attempted to write to the collector without first opening the file

<filenum-to-collector> input

INT:value

is the file number of the collector obtained through a call to the system OPEN procedure. The collector must be opened for waited I/O.

⟶

<level-3-buff> output

INT:ref:512

indicates that the spooler interface procedures are used to send data to the collector. The address of this buffer must be passed to the other interface procedures. This buffer is initialized as a result of this call.

<location> input

INT:ref:8

specifies a location for this job and overrides the location specified in the call to OPEN. This parameter must use the internal format for location specification:

```
<location>[0:3] := # group name blank-filled
          [4:7] := destination name blank-filled
```

The default location for the job is the location specified when the collector opened.

<form-name> input

INT:ref:8

specifies a form name for the job. The form name can be in letters, digits, or blanks. The default <form-name> is all blanks.

<report-name> input

INT:ref:8

specifies a report name for the job. The report name can be in letters, digits, or blanks.

The default <report-name> is the user name of the person executing the application program.

→

```
<num-of-copies>              input

  INT:value

  specifies the number of copies to print.  The default
  <num-of-copies> is 1.


<page-size>                  input

  INT:value

  specifies the page size used by PERUSE when a PAGE or LIST
  command is given.  The default <page-size> is 60.


<flags>                      input

  INT:value

  specifies certain attributes of the job.

  The bit fields are as follows:

    <flags>.<0:8>   = reserved for use by the collector
           .<9>     = HOLD flag:  0 = off
                                  1 = on
           .<10>    = HOLDAFTER flag:  0 = off
                                       1 = on
           .<11>    = SPOOLWRITE, SPOOLCONTROL, SPOOLCONTROLBUF,
                      and SPOOLSETMODE exit before writing the
                      level 3 buffer to the collector process, so
                      that user can checkpoint:  0 = no
                                                 1 = yes
           .<12>    = reserved for use by the collector
           .<13:15> = job priority

  The default is 4; job priority is 4; all other bits are set
  to 0.
```

⟶

<owner>	input
INT:value allows the caller to assign JOB ownership. The owner name is group number:user number. The default <owner> of a job is the user who opened the file to the collector.	

Condition Code Settings

The condition code has no meaning following a call to SPOOLSTART (see the <error-code> parameter).

Considerations

- SPOOLSTART--Bits <8:15>--Error on File to Collector

  Some file system errors have special significance to a process sending data to a collector; these errors are listed in the System Messages Manual.

  A program using level 1 or level 2 spooling gets these errors from the WRITE or OPEN procedures, while a program spooling at level 3 obtains these errors in bits <8:15> of a spooler error code in the %1000 range.

Example

```
CALL SPOOLSTART ( COLL^NUM , COLL^BUFFER );  ! start level 3.
```

Related Programming Manual

For programming information about the SPOOLSTART procedure, refer to the Spooler Programmer's Guide.

SPOOLWRITE PROCEDURE

The SPOOLWRITE procedure is used to write to a collector when the application process is spooling at level 3.

The SPOOLWRITE procedure compresses and blocks data into the level 3 buffer and, when the buffer is full, writes the buffer to the collector. This procedure must be used in place of the WRITE procedure, if a level 3 buffer is specified in a call to SPOOLSTART.

The syntax for SPOOLWRITE is:

```
<error-code> := SPOOLWRITE ( <level-3-buff>                    ! i, o
                           ,<print-line>                       ! i
                           ,<write-count>                      ! i
                           ,[ <bytes-written-to-buff> ] );     ! o


<error-code>                returned value

  INT

  returns one of the following spooler error codes:

              0 = successful operation
    %1000-%1377 = error on file to collector (<8:15> contains
                  a file system error code number; see
                  "Considerations")
          %10000 = missing parameter
          %10001 = parameter is present, but its content is wrong
          %11000 = checkpoint exit
          %11001 = attempted to write to the collector without
                   first opening the file


<level-3-buff>           input, output

  INT:ref:512

  is the <level-3-buff> specified in the call to SPOOLSTART.


<print-line>               input

  INT:ref:*

  is an array containing the line of data to be sent to the
  collector.  The size of <print-line> must not exceed 900 bytes.
```

⟶

<write-count> input

INT:value

is the number of bytes of <print-line> to be written. This must not exceed 900 bytes.

<bytes-written-to-buff> output

INT:ref:*

returns the number of bytes in the <level-3-buff> to be checkpointed.

Condition Code Settings

The condition code has no meaning following a call to SPOOLSTART (see the <error-code> parameter).

Considerations

- When <level-3-buff> Overflows

  Each call to SPOOLWRITE causes <print-line> to be written to the <level-3-buff>. When a call to SPOOLWRITE causes the <level-3-buff> to overflow, the buffer is written to the collector.

  The blocking and compression of data into the <level-3-buff> are invisible to the application process.

- SPOOLSTART and <flags>.<11>

  If bit 11 of the <flags> parameter of SPOOLSTART is set to 1, SPOOLWRITE exits with a spooler error code of %11000 prior to writing the <level-3-buff> to the collector. Applications running as a NonStop process pair can then perform a checkpoint, before the buffer is written to the collector. SPOOLWRITE should be recalled after checkpointing.

- SPOOLWRITE--Bits <8:15>--Error on File to Collector

  Some file system errors have special significance to a process sending data to a collector; these errors are listed in the System Messages Manual.

  A program using level 1 or level 2 spooling gets these errors from the WRITE or OPEN procedures, while a program spooling at level 3 obtains these errors in bits <8:15> of a spooler error code in the %1000 range.

Example

```
SPERRNUM := CALL SPOOLWRITE ( COLL^BUFFER , PRINT^LINE , LENGTH );
```

Related Programming Manual

For programming information about the SPOOLWRITE procedure, refer to the Spooler Programmer's Guide.

STEPMOM PROCEDURE

The STEPMOM procedure is called by a process when it wants to receive process STOP or ABEND messages for a process it did not create. Note that the caller of STEPMOM becomes the new "MOM" of the designated process. (That is, STEPMOM replaces the MOM field in the designated process's process control block with the process ID of its caller.) Therefore, only the caller receives the process deletion notification.

STEPMOM is typically used by the backup process of an nonnamed process pair to monitor its primary process. (This monitoring is automatic between members of named process pairs.)

The syntax for STEPMOM is:

```
CALL STEPMOM ( <process-id> );                    ! i

<process-id>                    input

  INT:ref:4

  is an array containing the PID of an already executing process,
  for which the calling process wants to receive the process
  deletion message.
```

Condition Code Settings

< (CCL) indicates that STEPMOM failed, or that no process designated <process-id> exists.

= (CCE) indicates that the caller is now the creator (MOM) of <process-id>.

> (CCG) does not return from STEPMOM.

Considerations

- Process Accessor ID and the Caller of STEPMOM

  The caller of STEPMOM must have the same process accessor ID as the process it is attempting to adopt, be the group manager of the process accessor ID, or super ID. Refer to the GUARDIAN Operating System Programmer's Guide for a description of "process accessor ID."

- Why STEPMOM Should Not Be Called for a NonStop Process Pair

  A process should not call STEPMOM for either member of a process pair. Adoption of a process pair by a third process causes errors and interferes with operation, since the operation depends upon each member of the process pair being the "MOM" of the other.

Figure 2-4 illustrates the effect of STEPMOM.

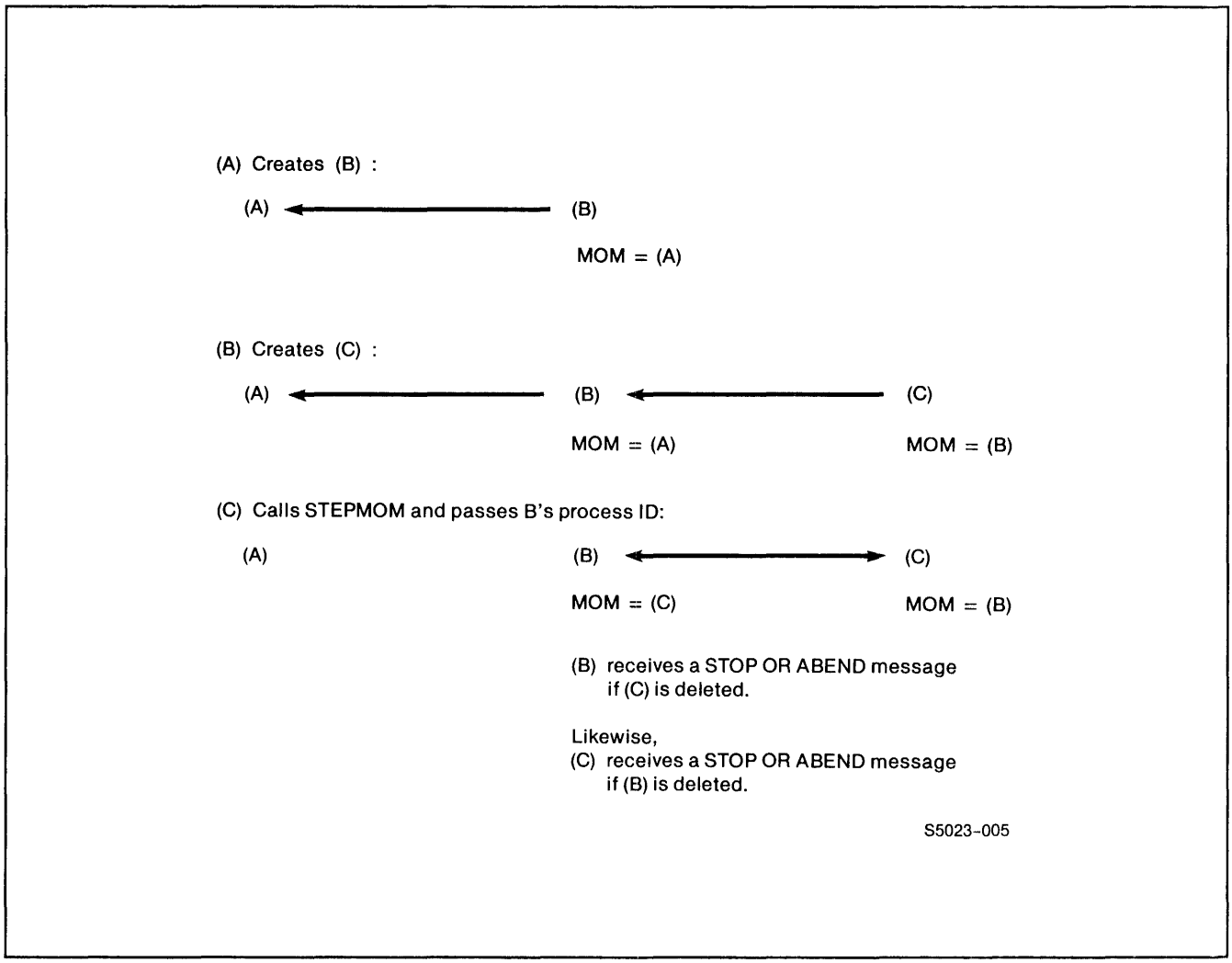


Figure 2-4. Effect of STEPMOM

Messages

- Process Normal Deletion Message

  The caller of STEPMOM receives system message (-5) if the <process-id> is being deleted normally because of a call to STOP.

- Process Abnormal Deletion Message

  The caller of STEPMOM receives system message (-6) if the <process-id> is being deleted abnormally because of a call to ABEND, or because the deleted process encountered a trap condition and was aborted by the operating system.

Example

```
CALL STEPMOM ( STEP^SON );
```

Related Programming Manual

For programming information about the STEPMOM process control procedure, refer to the GUARDIAN Operating System Programmer's Guide.

STOP PROCEDURE

The STOP procedure is used to delete a process or a process pair and to signal that the deletion was caused by a normal condition (that is, a STOP system message is sent to the deleted process's creator). STOP can be used by a process:

- To delete itself
- To delete its own backup

The caller of STOP must have the same process accessor ID as the process it is attempting to stop, be the group manager of the process accessor ID, or super ID. Refer to the GUARDIAN Operating System Programmer's Guide for a description of "process accessor ID."

When STOP executes, all open files associated with the deleted process(es) are automatically closed. If the process had BREAK enabled, BREAK is disabled.

The syntax for STOP is:

```
CALL STOP [ ( <process-id>                          ! i
              ,[ <stop-backup> ] );                 ! i
```

<process-id> input

INT:ref:4

is the process that is to be stopped, at this point, there are two choices:

- It can be omitted (or zero), meaning "stop myself."
- It is the process ID (PID) of the process to be stopped. If <process-id>[0:2] references a process pair and <process-id>[3] is specified as -1, then both members of the process pair are stopped.

<stop-backup> input

INT:value

if specified as 1, the current process's backup is stopped; otherwise, this parameter is ignored.

Condition Code Settings

< (CCL) indicates that the <process-id> parameter is invalid, or there was an error while stopping the process.

= (CCE) indicates that the STOP was successful.

> (CCG) does not return from STOP.

Considerations

- Differences Between STOP and ABEND Procedure

  Note that when used to stop the calling process, the STOP and ABEND procedures are identical in operation, with the exception of the particular system message sent to the deleted process's creator.

- Creator of the Process and the Caller of STOP

  If the caller of STOP is also the creator of the process being deleted, the caller receives the deletion message.

Message

- Process Normal Deletion Message

  The creator of the stopped process is sent a process normal deletion (STOP) system message (system message -5), indicating that the deletion occurred.

Examples

```
CALL STOP;              ! stop me.

CALL STOP ( PID );      ! stop the process with this PID.
```

Related Programming Manual

For programming information about the STOP process control procedure, refer to the GUARDIAN Operating System Programmer's Guide.

SUSPENDPROCESS PROCEDURE

The SUSPENDPROCESS procedure puts a process or process pair into the suspended state, preventing that process from being active (that is, executing instructions). (A process is removed from the suspended state and put back into the ready state if it is the object of a call to the ACTIVATEPROCESS procedure.)

The syntax for SUSPENDPROCESS is:

```
CALL SUSPENDPROCESS ( <process-id> );                    ! i

<process-id>                    input

  INT:ref:4

  is an array containing the process ID (PID) of the process to be
  suspended.  If <process-id>[0:2] references a process pair
  and <process-id>[3] is specified as -1, then both members of
  the process pair are suspended.
```

Condition Code Settings

<	(CCL)	indicates that SUSPENDPROCESS failed, or no process designated <process-id> exists.
=	(CCE)	indicates that <process-id> is suspended.
>	(CCG)	does not return from SUSPENDPROCESS.

Considerations

- Caller of SUSPENDPROCESS and Process Accessor ID

  The caller of SUSPENDPROCESS must have the same process accessor ID as the process or process pair it is attempting to suspend, or have a super ID, or be the group manager of the process accessor ID. Refer to the GUARDIAN Operating System Programmer's Guide for information about system security, specifically process accessor ID and super ID.

Example

```
CALL SUSPENDPROCESS ( PROG^ID );     ! suspend process.
```

Related Programming Manual

None

SYSTEMENTRYPOINTLABEL PROCEDURE

The SYSTEMENTRYPOINTLABEL procedure returns either the procedure label of the named entry point or, if not found, a zero.

The syntax for SYSTEMENTRYPOINTLABEL is:

```
<label> := SYSTEMENTRYPOINTLABEL ( <name>              ! i
                                 ,<len> );             ! i
```

<label> returned value

INT

is the procedure label.

<name> input

STRING:ref:*

is the entry point name and must be specified in upper-case letters.

<len> input

INT:value

is the length, in bytes, of <name>.

Condition Code Settings

The condition code has no meaning following a call to SYSTEMENTRYPOINTLABEL.

Example

```
EPNAME ':=' "FILEINFO";
EPLABEL := SYSTEMENTRYPOINTLABEL ( EPNAME , LEN );
```

Related Programming Manual

None

TIME PROCEDURE

The TIME procedure provides the current date and time in integer form.

The syntax for TIME is:

```
CALL TIME ( <date-and-time> );                ! o

<date-and-time>              output

  INT:ref:7

  returns an array with the current date and time in
  the following form:

    <date-and-time>[0] = year    (1983, 1984, ... )
                   [1] = month   (1-12)
                   [2] = day     (1-31)
                   [3] = hour    (0-23)
                   [4] = minute  (0-59)
                   [5] = second  (0-59)
                   [6] = .01 sec (0-99)
```

Condition Code Settings

The condition code has no meaning following a call to TIME.

Example

```
CALL TIME ( TIME^ARRAY );          ! TIME^ARRAY contains the
                                   ! integer form of current
                                   ! date and time.
```

Related Programming Manual

For programming information about the TIME utility procedure, refer to the GUARDIAN Operating System Programmer's Guide.

TIMESTAMP PROCEDURE

The TIMESTAMP procedure provides the internal form of the CPU interval clock where the application is running.

The syntax for TIMESTAMP is:

```
CALL TIMESTAMP ( <interval-clock> );                ! o

<interval-clock>              output

  INT:ref:3

  returns the current value of the interval clock in a three-word
  array.  A processor's interval clock is incremented every .01
  second.  <interval-clock> returns in the following form:

          -----------------------------------
    [0]  | most significant word             |
         |-----------------------------------|
    [1]  | interval clock                    |
         |-----------------------------------|
    [2]  | least significant word            |
         |-----------------------------------|
```

Condition Code Settings

The condition code has no meaning following a call to TIMESTAMP.

Considerations

- A timestamp is normally a 48-bit quantity equal to a number of of 10 millisecond units since 00:00, 0 January 1975.

Example

```
CALL TIMESTAMP ( TIMESTAMP^BUF );
```

Related Programming Manual

For programming information about the TIMESTAMP utility procedure, refer to the GUARDIAN Operating System Programmer's Guide.

TOSVERSION PROCEDURE

The TOSVERSION procedure provides an identifying letter and number indicating which version of the GUARDIAN operating system is running.

The syntax for TOSVERSION is:

```
<version> := TOSVERSION;

<version>                    returned value

  INT

  returns a value of the form:

    <0:7>   uppercase ASCII letter indicating system level:

              A      = T.O.S.
              B      = GUARDIAN
              C      = GUARDIAN / 1.1
              D      = GUARDIAN / EXPAND
              E      = GUARDIAN / EXPAND / Transaction Monitoring
                       Facility
              K, L   = GUARDIAN, NonStop system

    <8:15>  revision number of system in binary
```

Condition Code Settings

The condition code has no meaning following a call to TOSVERSION.

Example

```
VERSION := TOSVERSION;
```

Related Programming Manual

For programming information about the TOSVERSION utility procedure, refer to the GUARDIAN Operating System Programmer's Guide.

UNLOCKFILE PROCEDURE

The UNLOCKFILE procedure unlocks a disc file and any records in that file currently locked by the caller. Unlocking a file allows other processes to access the file. It has no effect on audited files.

The syntax for UNLOCKFILE is:

```
CALL UNLOCKFILE ( <filenum>                    ! i
                ,[ <tag> ]   );                ! i
```

<filenum> input

INT:value

is a number of an open file that identifies the file to be unlocked.

<tag> input

INT(32):value

is for nowait I/O only. <tag> is a value you define that uniquely identifies the operation associated with this UNLOCKFILE.

NOTE

The system stores the <tag> value until the I/O operation completes. The system then returns the <tag> information to the program in the <tag> parameter of the call to AWAITIO, thus indicating that the operation completed.

Condition Code Settings

< (CCL) indicates that an error occurred (call FILEINFO).

= (CCE) indicates that the UNLOCKFILE was successful.

> (CCG) indicates that the file is not a disc file.

Considerations

- Nowait and UNLOCKFILE

  The UNLOCKFILE procedure must complete with a corresponding call to the AWAITIO procedure when used with a file that is opened nowait.

- Locking Queue

  If any processes are queued in the locking queue for the file, the process at the head of the locking queue is granted access and is removed from the queue (the next READ or lock request moves to the head of the queue).

  If the next process in the locking queue is waiting to:

  --Lock the file or lock a record in the file, it is granted the lock (which excludes other processes from accessing the file) and resumes processing

  --To read the file, its READ is processed

- Transaction Monitoring Facility (TMF) and UNLOCKFILE

  Locks on files audited by TMF are released only when the transaction is ended or aborted by TMF; therefore, a locked file audited by TMF will be unlocked during an ENDTRANSACTION or ABORTTRANSACTION processing for that file. (Refer to the PATHWAY Programming Manual for information about TMF and locking files.)

Example

```
CALL UNLOCKFILE ( SAVE^FILENUM );
```

Related Programming Manual

For programming information about the UNLOCKFILE file system procedure, refer to the ENSCRIBE Programming Manual and the GUARDIAN Operating System Programmer's Guide.

UNLOCKREC PROCEDURE

The UNLOCKREC procedure unlocks a record currently locked by the caller. UNLOCKREC unlocks the record at the current position, allowing other processes to access that record. UNLOCKREC has no effect on audited files.

The syntax for UNLOCKREC is:

```
CALL  UNLOCKREC ( <filenum>                    ! i
                ,[ <tag> ] );                  ! i
```

<filenum> input

INT:value

is the number of an open file that identifies the file containing the record to be unlocked.

<tag> input

INT(32):value

is for nowait I/O only. <tag> is a value you define that uniquely identifies the operation associated with this UNLOCKREC.

NOTE

The system stores this <tag> value until the I/O operation completes. The system then returns the <tag> information to the program in the <tag> parameter of the call to AWAITIO, thus indicating that the operation completed.

Condition Code Settings

< (CCL) indicates that an error occurred (call FILEINFO).

= (CCE) indicates that the UNLOCKREC is successful.

> (CCG) indicates that the file is not a disc file.

Considerations

- File-Opened Nowait and UNLOCKREC

  The UNLOCKREC procedure must complete with a corresponding call to the AWAITIO procedure when used with a file that is opened nowait.

- Queuing Processes and UNLOCKREC

  If any processes are queued in the locking queue for the record, the process at the head of the locking queue is granted access and is removed from the queue (the next READ or lock request moves to the head of the queue).

  If the process granted access is waiting to lock the record, it is granted the lock (which excludes other process from accessing the record) and resumes processing.

  If the process granted access is waiting to read the record, its READ is processed.

- Calling UNLOCKREC After KEYPOSITION

  If the call to UNLOCKREC immediately follows a call to KEYPOSITION where a nonunique alternate key is specified, the UNLOCKREC fails. A subsequent call to FILEINFO returns file system error 46 (invalid key). However, if an intermediate call to READ or READLOCK is performed, the call to UNLOCKREC is permitted.

- Unlocking Several Records

  If several records need to be unlocked, the UNLOCKFILE procedure can be called to unlock all records currently locked by the caller (rather than unlocking the records through individual calls to UNLOCKREC).

- Current-State Indicators After UNLOCKREC

  For key-sequenced, relative, and entry-sequenced files, the current-state indicators after an UNLOCKREC remain unchanged.

- File Pointers After UNLOCKREC

  For unstructured files, the current-record pointer and the next-record pointer remain unchanged.

- Transaction Monitoring Facility (TMF) and UNLOCKREC

  A record that is locked in a file audited by TMF is unlocked when an ABORTTRANSACTION or ENDTRANSACTION procedure is called for that file. Locks on files audited by TMF are released only when the transaction is ended or aborted by TMF. (Refer to the PATHWAY Programming Manual for additional information about record locking and TMF.)

Example

```
CALL UNLOCKREC ( FILE^NUM );
```

Related Programming Manual

For programming information about the UNLOCKREC file system procedure, refer to the ENSCRIBE Programming Manual.

USERIDTOUSERNAME PROCEDURE

The USERIDTOUSERNAME procedure returns the user name, from the file $SYSTEM.SYSTEM.USERID, that is associated with a designated user ID.

The syntax for USERIDTOUSERNAME is:

```
CALL USERIDTOUSERNAME ( <id-name> );              ! i, o

<id-name>                      input, output

  INT:ref:8

  on input, contains the user ID to be converted to a user
  name.  The user ID is passed in the form:

    <id-name>.<0:7>     = group ID {0:255}
             .<8:15>    = user ID  {0:255}

  on the return, contains the user name associated with the
  specified user ID in the form:

    <id-name>     FOR 4 = group name, blank-filled
    <id-name>[4] FOR 4 = user name, blank-filled
```

Condition Code Settings

< (CCL) indicates that <id-name> is out of bounds or that an I/O error occurred with the $SYSTEM.SYSTEM.USERID file.

= (CCE) indicates that the designated user name returned.

> (CCG) indicates that the specified user ID is undefined.

Example

```
CALL USERIDTOUSERNAME ( ID^NAME );
```

Related Programming Manual

For programming information about the USERIDTOUSERNAME security procedure, refer to the GUARDIAN Operating System Programmer's Guide.

USERNAMETOUSERID PROCEDURE

The USERNAMETOUSERID procedure returns the user ID, from the file $SYSTEM.SYSTEM.USERID, that is associated with a designated user name.

The syntax for USERNAMETOUSERID is:

```
CALL USERNAMETOUSERID ( <name-id> );                ! i, o


<name-id>                       input, output

  INT:ref:8

  on input, contains the user name to be converted to a
  user ID.  The user name is passed in the form:

    <name-id>    FOR 4 = group name, blank-filled
    <name-id>[4] FOR 4 = user name, blank-filled

  on the return, contains the user ID associated with the
  specified user name in the form:

    <name-id>.<0:7>    = group ID {0:255}
    <name-id>.<8:15>   = user ID  {0:255}
```

Condition Code Settings

< (CCL) indicates that <name-id> is out of bounds or that an I/O error occurred with the $SYSTEM.SYSTEM.USERID file.

= (CCE) indicates that the designated user ID returned.

> (CCG) indicates that the specified user name is undefined.

Example

```
CALL USERNAMETOUSERID ( NAME^ID );
```

Related Programming Manual

For programming information about the USERNAMETOUSERID security procedure, refer to the GUARDIAN Operating System Programmer's Guide.

USESEGMENT PROCEDURE

The USESEGMENT procedure selects a particular extended data segment to be currently addressable by the calling process.

The syntax for USESEGMENT is:

```
<old-segment-id> := USESEGMENT ( <segment-id> );        ! i
```

<old-segment-id> returned value

INT

returns the segment ID of the previously used segment if any; otherwise, it is -1.

WARNING

<old-segment-id> should be a simple INT(32) variable; otherwise, the assignment can alter the condition code.

<segment-id> input

INT:value

if present, is the segment ID of the segment used or -1 if no segment is used. If this parameter is not supplied, the current segment remains unchanged.

Condition Code Settings

<	(CCL)	indicates that <segment-id> is not allocated, or that the segment cannot be used by a nonprivileged caller.
=	(CCE)	indicates that the operation is successful.
>	(CCG)	does not return from USESEGMENT.

Considerations

- Because segment relocation is done, the first byte of any extended segment has the address %2000000D.

- Only one extended data segment can be in use at any given time.

Example

```
OLD^SEG^ID := USESEGMENT ( NEW^SEG^ID );  ! change segments.
```

Related Programming Manual

None

VERIFYUSER PROCEDURE

VERIFYUSER allows a process to assume a user's identity by using the given user ID (see "Condition Code Settings").

The syntax for VERIFYUSER is:

```
CALL VERIFYUSER ( <user-name-or-id>                          ! i
                ,[ <logon> ]                                 ! i
                ,[ <default> , <default-len> ]  );           ! o, i
```

<user-name-or-id> input

INT:ref:12

is an array containing the name or user ID of the user to be verified or logged on, as follows:

```
  <user-name-or-id>[0:3] = group name, blank-filled
  <user-name-or-id>[4:7] = user name, blank-filled
or
  <user-name-or-id>[0].<0:7>  = group ID
  <user-name-or-id>[0].<8:15> = user ID
  <user-name-or-id>[1:7]      = zeros (ASCII nulls)
```

In either case:

```
  <user-name-or-id>[8:11] = password (if supplied)
                            blank-filled
```

<logon> input

INT:value

if present, has the following meaning:

```
     0 : verify user but do not log on
  <> 0 : verify user and log on
```

if omitted, is assumed to have a value of 0.

→

```
<default>                       output

  INT:ref:18

  if present, returns information regarding the user specified
  in <user-name-or-id>:

    <default>[0:3]          = group name, blank-filled
             [4:7]          = user name, blank-filled
             [8].<0:7>      = group ID
                .<8:15>     = user ID
             [9:12]         = default volume, blank-filled
             [13:16]        = default subvolume, blank-filled

             [17].<0:15>    = default file security, as follows:
                              .<0:3>   = unused
                              .<4:6>   = read
                              .<7:9>   = write
                              .<10:12> = execute
                              .<13:15> = purge

                                  where  0 = A
                                         1 = G
                                         2 = O
                                         4 = N
                                         5 = C
                                         6 = U
                                         7 = -


<default-len>                   input

  INT:value

  is the length, in bytes, of the <default> array. <default-len>
  is required if <default> is specified.  This number should
  always be specified as 36; in the future, new fields can be
  added to <default>, requiring <default-len> to become larger.
```

Condition Code Settings

< (CCL) indicates that a buffer is out of bounds, or that an I/O error occurred on the user ID file ($SYSTEM.SYSTEM.USERID).

= (CCE) indicates a successful verification and/or logon.

NOTE

Condition code CCE returns under the following conditions:

- Specifying 0 for the <logon> parameter verifies that there is a user with that name on the system, but you cannot assume that user's identity and you cannot log on.

- If the <logon> parameter is a value other than 0, you can assume that user's ID if:

  --You are super.super ID (255,255).
  --You are the group manager (*,255).
  --You know the user's password.

  If you assume one of the above IDs, then your process access ID and creator access ID changes, you become a local user, and your default file security changes to what is established in the local USERID file.

> (CCG) indicates that there is no such user or that the password is invalid.

Considerations

- Successful Logon With This Procedure

  Following a successful logon with this procedure, the calling process is considered local with respect to the system on which it is running.

- A process that passes an invalid password to VERIFYUSER for the third time is suspended for 60 seconds.

- System Users

  System users are defined through the COMINT ADDUSER command. All COMINT commands are described in the GUARDIAN Operating System Utilities Reference Manual.

Example

```
USER := 3 '<<' 8 + 17;                  ! user ID 3,17.
USER[1] ':=' 0 & USER[1] FOR 6;         ! all zeros.
USER[8] ':=' password FOR 8;
LOGON := 1;                             ! log this user on.

CALL VERIFYUSER ( USER , LOGON , DEFAULT , DEFAULT^LEN );

IF < THEN ...                           ! buffer or I/O error,
ELSE IF > THEN ...                      ! no such user, or bad
                                        ! password.
ELSE ...                                ! successful.
      .
```

The array "USER" is prepared with the user and group ID and then passed to VERIFYUSER. VERIFYUSER logs on the process with the user ID 17 and group ID 3.

Related Programming Manual

For programming information about the VERIFYUSER security procedure, refer to the GUARDIAN Operating System Programmer's Guide.

WRITE PROCEDURE

The WRITE procedure is used to output data from an array in the application program to an open file (see "Considerations").

The syntax for WRITE is:

```
CALL WRITE ( <filenum>                          ! i
           ,<buffer>                            ! i
           ,<write-count>                       ! i
           ,[ <count-written> ]                 ! o
           ,[ <tag> ]  );                       ! i
```

<filenum> input

INT:value

is the number of an open file that identifies the file to be written.

<buffer> input

INT:ref:*

is an array containing the information to be written to the file.

<write-count> input

INT:value

is the number of bytes to be written:

{0:4096} for disc files
{0:32767} for nondisc files
{0:32000} for interprocess files
{0:102} for the operator console

For key-sequenced and relative files, 0 is illegal. For entry-sequenced files, 0 indicates an empty record.

→

<count-written> output

INT:ref:1

is for wait I/O only. <count-written> returns a count of the number of bytes written to the file.

<tag> input

INT(32):value

is for nowait I/O only. <tag> is a value you define that uniquely identifies the operation associated with this WRITE.

NOTE

The system stores this <tag> value until the I/O operation completes. The system then returns the <tag> information to the program in the <tag> parameter of the call to AWAITIO, thus indicating that the operation completed.

Condition Code Settings

< (CCL) indicates that an error occurred (call FILEINFO).

= (CCE) indicates that the WRITE is successful.

> (CCG) does not return from WRITE.

Considerations

- Waited WRITE

  If a waited WRITE is executed, the <count-written> parameter indicates the number of bytes actually written.

- Nowait WRITE

  If a nowait WRITE is executed, <count-written> has no meaning and can be omitted. The count of the number of bytes written is obtained when the I/O operation completes through the <count-transferred> parameter of the AWAITIO procedure.

  The WRITE procedure must complete with a corresponding call to the AWAITIO procedure when used with a file that is opened nowait.

Disc File Considerations

- File Is Locked

  If a call to WRITE is made and the file is locked through a file number other than that supplied in the call, the call is rejected with file system error 73 ("file is locked").

- Inserting a New Record Into a File

  The WRITE procedure inserts a new record into a file in the position designated by the file's primary key:

Key-Sequenced Files	The record is inserted in the position indicated by the value in its primary-key field.
Relative Files	After an OPEN or an explicit positioning by its primary key, the record is inserted in the designated position. Subsequent WRITEs without intermediate positioning insert records in successive record positions. If -2D is specified in a preceding positioning, the record is inserted in an available record position in the file. If -1D is specified in a preceding positioning, the record is inserted following the last position used in the file. There does not have to be an existing record in that position at the time of the WRITE.
Entry-Sequenced Files	The record is inserted following the last record currently existing in the file.

  NOTE

  If the insert is to be made to a key-sequenced or relative file and the record already exists, the WRITE fails, and a subsequent call to FILEINFO returns file system error 10.

Unstructured Files	The record is inserted at the position indicated by the current value of the next-record pointer.

- Structured Files

  --Inserting records into relative and entry-sequenced files

  If the insertion is to a relative or entry-sequenced file, the file must be positioned currently through its primary key. Otherwise, the WRITE fails, and a subsequent call to FILEINFO returns file system error 46 ("invalid key specified").

--Current-state indicators after WRITE

After a successful WRITE, the current-state indicators for positioning mode and comparison length remain unchanged.

For key-sequenced files, the current position and the current primary-key value remain unchanged.

For relative and entry-sequenced files, the current position is that of the record just inserted and the current primary-key value is set to the value of the record's primary key.

- Unstructured Files

--Unstructured WRITEs

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

--The number of bytes written

If an unstructured file is created with the ODDUNSTR (odd unstructured file) parameter set, the number of bytes written is exactly the number of bytes specified with <write-count>. If the ODDUNSTR parameter is not set when the file is created, the value of <write-count> is rounded up to an even number before the WRITE is executed.

The ODDUNSTR parameter is set when the file is created, either with <filetype>.<12> of the CREATE procedure or with the Peripheral Utilities Program SET and CREATE commands.

--File pointers after WRITE

After a successful WRITE to an unstructured file, the file pointers have these values:

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

--WRITE to an even unstructured disc file

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

--Block size

No block can span more than two extents.

Interprocess Communication Consideration

- Indication That the Destination Process Is Running

  If the WRITE is to another process, successful completion of the WRITE (or AWAITIO if nowait) indicates that the destination process is running.

Example

```
CALL WRITE ( OUT^FILE , OUT^BUFFER , 72 );
```

Related Programming Manuals

For programming information about the WRITE file system procedure, refer to the GUARDIAN Operating System Programmer's Guide, the ENSCRIBE Programming Manual, and the data communication manuals.

WRITEREAD PROCEDURE

The WRITEREAD procedure writes data to a file from an array in the application process, then waits for data to be transferred back from the file. The data returns in the same array used for the WRITE.

Terminals	A special hardware feature is incorporated in the asynchronous multiplexer controller, which ensures that the system is ready to read from the terminal as soon as the WRITE is completed.
Interprocess Communication	The WRITEREAD procedure is used to originate a message to another process which was previously opened, then waits for a reply from that process.

The syntax for WRITEREAD is:

```
CALL WRITEREAD ( <filenum>                          ! i
                ,<buffer>                           ! i,o
                ,<write-count>                      ! i
                ,<read-count>                       ! i
                ,[ <count-read> ]                   ! o
                ,[ <tag> ]  );                      ! i
```

<filenum> input

INT:value

is the number of an open file that identifies the file where the WRITE/READ is to occur.

<buffer> input, output

INT:ref:*

is an array containing information to be written to the file.

On return, <buffer> contains the information read from the file.

→

<write-count> input

INT:value

is the number of bytes to be written:

{0:32767} for terminals
{0:32000} for interprocess files

NOTE

When using terminals in block mode, an error 21 occurs if <write-count> exceeds 256 bytes.

<read-count> output

INT:value

returns the number of bytes to be read:

{0:32767} for terminals
{0:32000} for interprocess files

NOTE

When using terminals in block mode, an error 21 occurs if <read-count> exceeds 256 bytes.

<count-read> output

INT:ref:1

is for wait I/O only. It returns a count of the number of bytes returned from the file into <buffer>.

<tag> input

INT(32):value

is for nowait I/O only. <tag> is a value you define that uniquely identifies the operation associated with this WRITEREAD.

NOTE

The system stores this <tag> value until the I/O operation completes. The system then returns the <tag> information to the program in the <tag> parameter of the call to AWAITIO, thus indicating that the operation completed.

Condition Code Settings

< (CCL) indicates that an error occurred (Call FILEINFO).

= (CCE) indicates the the WRITEREAD is successful.

> (CCG) indicates that CNTRL-Y is pressed on the terminal.

Considerations

- Waited READ

  If a waited READ is executed, the <count-read> parameter indicates the number of bytes actually read.

- Nowait READ

  If a nowait READ is executed, <count-read> has no meaning and can be omitted. The count of the number of bytes read is obtained when the I/O operation completes through the <count-transferred> parameter of the AWAITIO procedure.

  The WRITEREAD procedure must complete with a corresponding call to the AWAITIO procedure when used with a file that is opened nowait.

- Carriage Return/Line Feed Sequence After WRITE

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

Example

```
CALL WRITEREAD ( FILE^NUM , INOUT^BUFFER , 1 , 72 , NUM^READ );
```

The INOUT^BUFFER contains the information to be written, and after the WRITE it contains information that was read. In this case, 1 byte is to be written, and 72 bytes are to be read. NUM^READ indicates how many bytes are read into the INOUT^BUFFER.

Related Programming Manuals

For programming information about the WRITEREAD file system procedure, refer to the GUARDIAN Operating System Programmer's Guide, the ENSCRIBE Programming Manual, and the data communication manuals.

WRITEUPDATE PROCEDURE

The WRITEUPDATE procedure transfers data from an array in the application program to a file.

- For disc files, WRITEUPDATE has two functions:

  1. To alter the contents of the record at the current position

  2. To delete the record at the current position in a key-sequenced or relative file.

WRITEUPDATE is used for processing data at random. Data from the application process's array is written in the position indicated by the setting of the current-record pointer. A call to this procedure typically follows a corresponding call to the READ or READUPDATE procedure. The current-record and next-record pointers are not affected by the WRITEUPDATE procedure.

- For magnetic tapes, WRITEUPDATE is used to replace a record in the middle of an already written tape. The tape is backspaced one record; the data from the application process's array is written in that area.

The syntax for WRITEUPDATE is:

```
CALL WRITEUPDATE ( <filenum>                          ! i
                  ,<buffer>                           ! i
                  ,<write-count>                      ! i
                  ,[ <count-written> ]                ! o
                  ,[ <tag> ]  );                      ! i
```

<filenum> input

INT:value

is a number of an open file that identifies the file to be written.

<buffer> input

INT:ref:*

is an array containing the information to be written to the file.

→

<write-count> input

INT:value

is the number of bytes to be written to the file:

{0:4096} for disc files
{0:32767} for magnetic tapes

Key-sequenced and relative files = 0 means delete the record.
Entry-sequenced files = 0 anything <> the record's length is illegal.

<count-written> output

INT:ref:1

is for wait I/O only. It returns a count of the number of bytes written to the file.

<tag> input

INT(32):value

is for nowait I/O only. <tag> is a value you define that uniquely identifies the operation associated with this WRITEUPDATE.

NOTE

The system stores this <tag> value until the I/O operation completes. The system returns the <tag> information back to the program in the <tag> parameter of the call to AWAITIO, thus indicating that the operation completed.

Condition Code Settings

< (CCL) indicates that an error occurred (call FILEINFO).

= (CCE) indicates the the WRITEUPDATE was successful.

> (CCG) does not return from WRITEUPDATE.

Considerations

- Bad I/O Count

  For DP1 files:

  If the <write-count> parameter attempts to transfer too much data, the call is rejected with file system error 21. No single disc transfer can span more than two extents.

  For DP2 files:

  There is no restriction on the beginning file position. The BUFFERSIZE attribute value (which is set by specifying SETMODE function 93) does not constrain the allowable <write-count> in any way; however, there is a performance penalty if the WRITE does not start on a BUFFERSIZE boundary and have a <write-count> of <= BUFFERSIZE. DP2 executes your requested I/O in units (possibly multiple) of BUFFERSIZE blocks, starting on a block boundary.

- Deleting Locked Records

  Deletion of a locked record implicitly unlocks that record.

- Waited WRITEUPDATE

  If a waited WRITEUPDATE is executed, the <count-written> parameter indicates the number of bytes actually written.

- Nowait WRITEUPDATE and Transaction Monitoring Facility (TMF)

  If a nowait WRITEUPDATE is executed, <count-written> has no meaning and can be omitted. The count of the number of bytes written is obtained through the <count-transferred> parameter of the AWAITIO procedure when the I/O completes.

  The WRITEUPDATE procedure must complete with a corresponding call to the AWAITIO procedure when used with a file that is opened nowait. For files audited by TMF, the AWAITIO procedure must be called before the ENDTRANSACTION or ABORTTRANSACTION procedure is called.

Disc File Considerations

- Random Processing and WRITEUPDATE

  For key-sequenced, relative, and entry-sequenced files, random processing implies that a designated record must exist. This means that positioning for WRITEUPDATE is always to the record described by the exact value of the current key and current-key specifier. If such a record does not exist, the call to WRITEUPDATE is rejected with file system error 11 ("record does not exist").

- Record Does Not Exist

  The positioning for WRITEUPDATE is always to the record described by the exact value of the current key and current-key specifier. Therefore, if such a record does not exist, the call to WRITEUPDATE is rejected with file system error 11.

- File Is Locked

  If a call to WRITEUPDATE is made and the file is locked through a file number other than that supplied in the call, the call is rejected with file system error 73 ("file is locked").

- When the Just Read Record Is Updated

  A call to WRITEUPDATE following a call to READ, without intermediate positioning, updates the record just read.

- A WRITE to an Even Unstructured Disc File

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

- Unstructured Files

  --Unstructured disc file: transferring data

    If the WRITE is to an unstructured disc file, data is transferred to the record location specified by the current-record pointer.

  --File pointers after a successful WRITEUPDATE

    After a successful WRITEUPDATE to an unstructured file, the current-record and next-record pointers are unchanged.

  --The number of bytes written

    If the unstructured file is created with the ODDUNSTR (odd unstructured file) parameter set, the number of bytes written is exactly the number of bytes specified with <write-count>. If the ODDUNSTR parameter is not set when the file is created, the value of <write-count> is rounded up to an even number before the WRITEUPDATE is executed.

    The ODDUNSTR parameter is set when the file is created, either with <filetype>.<12> of the CREATE procedure or with the Peripheral Utilities Program SET and CREATE commands.

- Structured Files

  --Calling WRITEUPDATE after KEYPOSITION

  If the call to WRITEUPDATE immediately follows a call to KEYPOSITION where a nonunique alternate key is specified as the access path, the WRITEUPDATE fails. A subsequent call to FILEINFO returns file system error 46 ("invalid key"). However, if an intermediate call to READ or READLOCK is performed, the call to WRITEUPDATE is permitted because a unique record is identified.

  --Specifying <write-count> for entry-sequenced files

  For entry-sequenced files, the value of <write-count> must match exactly the <write-count> value specified when the record was originally inserted into the file.

  --Changing the <primary-key> of a key-sequenced record

  An update to a record in a key-sequenced file cannot alter the value of the <primary-key> field. Changing the <primary-key> field must be done by deleting the old record (WRITEUPDATE with <write-count> = 0) and inserting a new record with the key field changed (WRITE).

  --Current-state indicators after WRITEUPDATE

  After a successful WRITEUPDATE, the current-state indicators remain unchanged.

Magnetic Tape Considerations

- WRITEUPDATE is not permitted on the 5106 Tri-Density Tape Drive.
- Specifying the Correct Number of Bytes Written

  When WRITEUPDATE is used with magnetic tape the number of bytes to be written must fit exactly; otherwise, information on the tape can be lost. However, no error indication is given.

- Limitation of WRITEUPDATE to the Same Record

  Five is the maximum number of times a WRITEUPDATE can be executed to the same record on tape.

Example

```
CALL WRITEUPDATE ( TAPE^NUM , TAPE^BUF , NUM^READ , NUM^WRITTEN  );
```

The application makes the necessary changes to the record in TAPE^BUF, then edits the tape by calling WRITEUPDATE. The tape is backspaced over the record just read, then updated by writing the new record in its place. NUM^READ indicates the number of bytes to be written (ensuring that the same number of bytes just read are also written).

Related Programming Manuals

For programming information about the WRITEUPDATE file system procedure, refer to the ENSCRIBE Programming Manual.

WRITEUPDATEUNLOCK PROCEDURE

The WRITEUPDATEUNLOCK procedure performs random processing of records in a disc file. WRITEUPDATEUNLOCK has two functions:

1. To alter, then unlock, the record's contents at the current position

2. To delete the record at the current position in a key-sequenced or relative file.

A call to WRITEUPDATEUNLOCK is equivalent to a call to WRITEUPDATE followed by a call to UNLOCKREC. However, the WRITEUPDATEUNLOCK procedure requires less system processing than do the separate calls to WRITEUPDATE and UNLOCKREC.

The syntax for WRITEUPDATEUNLOCK is:

```
CALL WRITEUPDATEUNLOCK ( <filenum>                          ! i
                        ,<buffer>                           ! i
                        ,<write-count>                      ! i
                        ,[ <count-written> ]                ! o
                        ,[ <tag> ]  );                      ! i
```

<filenum> input

INT:value

is a number of an open file that identifies the file to be written.

<buffer> input

INT:ref:*

is an array containing the data to be written to the file.

<write-count> input

INT:value

is the number of bytes to be written to the file: {0:4096}.

Key-sequenced and relative files = 0 delete the record
Entry-sequenced files = 0 is illegal (error 21)

→

<count-written> output

INT:ref:1

is for wait I/O only. It returns an integer indicating the number of bytes written to the file.

<tag> input

INT(32):value

is for nowait I/O only. <tag> is a value you define that uniquely identifies the operation associated with this WRITEUPDATEUNLOCK.

NOTE

The system stores this <tag> value until the I/O operation completes. The system then returns the <tag> information to the program in the <tag> parameter of the call to AWAITIO, thus indicating that the operation completed.

Condition Code Settings

< (CCL) indicates that an error occurred (call FILEINFO).

= (CCE) indicates that the WRITEUPDATEUNLOCK was successful.

> (CCG) does not return from WRITEUPDATEUNLOCK.

Considerations

- Nowait and WRITEUPDATEUNLOCK

  The WRITEUPDATEUNLOCK procedure must complete with a corresponding call to the AWAITIO procedure when used with a file that is opened nowait. For files audited by the Transaction Monitoring Facility, the AWAITIO procedure must be called to complete the WRITEUPDATELOCK operation before ENDTRANSACTION or ABORTTRANSACTION is called.

- Random Processing and WRITEUPDATEUNLOCK

  For key-sequenced, relative, and entry-sequenced files, random processing implies that a designated record must exist. This means positioning for WRITEUPDATEUNLOCK is always to the record described by the exact value of the current key and current-key specifier. If such a record does not exist, the call to WRITEUPDATEUNLOCK is rejected with file system error 11 ("record does not exist").

- Unstructured Files--Pointers Unchanged

  For unstructured files, data is written in the position indicated by the current-record pointer. A call to WRITEUPDATEUNLOCK for an unstructured file typically follows a call to POSITION or READUPDATE. The current-record and next-record pointers are not changed by a call to WRITEUPDATEUNLOCK.

- How WRITEUPDATEUNLOCK Works

  The record unlocking performed by WRITEUPDATEUNLOCK functions in the same manner as UNLOCKREC.

- Record Does Not Exist

  Positioning for WRITEUPDATEUNLOCK is always to the record described by the exact value of the current key and current-key specifier. Therefore, if such a record does not exist, the call to WRITEUPDATEUNLOCK is rejected with file system error 11.

- See the "Considerations" for WRITEUPDATE.

Example

```
CALL WRITEUPDATEUNLOCK ( OUT^FILE , OUT^BUFFER , 72   &
                       , NUM^WRITTEN );
```

Related Programming Manuals

For programming information about the WRITEUPDATEUNLOCK file system procedure, refer to the ENSCRIBE Programming Manual.

SECTION 3

SEQUENTIAL I/O PROCEDURES

The sequential I/O (SIO) procedures provide Tandem Application Language (TAL) programmers with a small, standardized set of procedures for performing sequential input and output operations to files.

The SIO procedures are recommended for someone who writes programs that are similiar to Tandem subsystems requiring sequential access to files. If you need a special file capability use the file system procedure calls instead.

Generally, you should not use these procedures and other GUARDIAN operating system I/O procedures together on the same file.

The source file named $SYSTEM.SYSTEM.GPLDEFS is used with the SIO procedures. It provides the TAL definitions for allocating control block space, for assigning open characteristics to the file, and for altering and checking the file transfer characteristics. 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.

This section describes each SIO procedure in detail. The SIO procedures are listed alphabetically for easier reference.

CHECK^BREAK PROCEDURE

The CHECK^BREAK procedure tests whether the BREAK key has been typed since the last CHECK^BREAK.

The syntax for CHECK^BREAK is:

```
<state> := CHECK^BREAK ( { <common-fcb> } );          ! i
                         { <file-fcb>   }             ! i
```

<state> returned value

INT

returns a value indicating whether or not the BREAK key has been typed. Values are:

1 = BREAK key typed; the process owns BREAK.
0 = BREAK key not typed; or this process does not own BREAK.

<common-fcb> input

INT:ref:*

identifies the file to be checked for BREAK. <common-FCB> is allowed for convenience.

<file-fcb> input

INT:ref:*

identifies the file to be checked for BREAK.

Condition Code Settings

The condition code has no meaning following a call to CHECK^BREAK.

Considerations

- Default Action

  If a carriage return/line feed (CR/LF) on BREAK is enabled (that is, BREAK ownership is taken by the process), the CR/LF default case sequence is executed on the terminal where BREAK is typed.

- For information about terminals, refer to the GUARDIAN Operating System Programmer's Guide.

Example

```
BREAK := CHECK^BREAK ( OUT^FILE );
```

Related Programming Manual

For programming information about the CHECK^BREAK procedure, refer to the GUARDIAN Operating System Programmer's Guide.

CHECK^FILE PROCEDURE

The CHECK^FILE procedure checks the file characteristics.

The syntax for CHECK^FILE is:

```
<retval> := CHECK^FILE ( { <common-fcb> }                    ! i
                         { <file-fcb>   }                    ! i

  ,<operation> );                                            ! i
```

<retval> returned value

INT

returns a value for the requested operation (see Table 3-1 for these values).

<common-fcb> or <file-fcb> input

INT:ref:*

identifies which file is checked. The common file control block (FCB) can be used for certain types of checks; the common FCB must be used for the checks FILE^BREAKHIT, FILE^ERRORFILE, and FILE^TRACEBACK. Specifying an improper FCB causes an error indication.

<operation> input

INT:value

specifies which file characteristic is checked. The <operation>s and their associated <retval>s are listed in Table 3-1.

Condition Code Settings

The condition code has no meaning following a call to CHECK^FILE.

Considerations

- During the execution of this procedure, the detection of any error causes the display of an error message, and the process is aborted.

- In Table 3-1, the column labeled "State of the File" is flagged with the following:

Open	The file must be open to obtain certain characteristics of the specified file.
Blank	This indicates that the file can be either open or closed.

Table 3-1. CHECK^FILE Operations

⟨operation⟩	State of File	⟨retval⟩
FILE∧ABORT∧XFERERR	Open	0 if the process is not to abort upon detection of a fatal error in the file. 1 if the process is to abort.
FILE∧ASSIGNMASK1		⟨high-order word of ASSIGN fieldmask⟩ returns the high-order word of the ASSIGN message field mask in the FCB. This value generally has meaning only after being set by the INITIALIZER procedure.
FILE∧ASSIGNMASK2		⟨low-order word of ASSIGN fieldmask⟩ returns the low-order word of the ASSIGN message field mask in the FCB. This value generally has meaning only after being set by the INITIALIZER procedure.
FILE∧BLOCKBUFLEN		⟨block buffer length⟩ returns a count of the number of bytes used for blocking.
FILE∧BREAKHIT		⟨state of the break hit bit⟩ 0 if the break hit bit is equal to 0 in the FCB. 1 if the break hit bit is equal to 1 in the FCB. The break hit bit is an internal indicator normally used only by the SIO procedures.
NOTE When using the break handling procedures, do not use FILE∧BREAKHIT to determine if the BREAK key has been typed. Instead, the CHECK∧BREAK procedure must be called.		
FILE∧BWDLINKFCB		⟨backward link pointer⟩ returns the address of the FCB pointed to by the backward link pointer within the FCB. This indicates the linked-to FCBs that need to be checkpointed after an OPEN∧FILE or CLOSE∧FILE.
FILE∧CHECKSUM		⟨checksum word⟩ returns the value of the checksum word in the FCB.
FILE∧CREATED	Open	⟨state of the created bit⟩ 0 if a file was not created by OPEN∧FILE. 1 if a file was created by OPEN∧FILE.
FILE∧COUNTXFERRED	Open	⟨count transferred⟩ returns a count of the number of bytes transferred in the latest physical I/O operation.
FILE∧CRLF∧BREAK	Open	⟨state of CR/LF break bit⟩ 0 if no CR/LF sequence is to be issued to the terminal upon break detection. 1 if this sequence is to be issued.

Table 3-1. CHECK^FILE Operations (Continued)

⟨operation⟩	State of File	⟨retval⟩
FILE^DUPFILE	Open	@⟨dupfile fcb⟩ returns the word address of the duplicate file FCB. A 0 is returned if there is no duplicate file.
FILE^ERROR	Open	⟨error⟩ returns the error number of the latest error that occurred within the file.
FILE^ERRORFILE		@⟨error file fcb⟩ returns the word address within the FCB of the reporting error file. A 0 is returned if there is none.
FILE^ERROR^ADDR		@⟨error⟩ returns the word address within the FCB where the error code is stored.
FILE^FCB^ADDR		returns the address of the file control block.
FILE^FILEINFO	Open	⟨file info⟩ ⟨file info⟩.⟨0:3⟩ = file type: 0 = unstructured 1 = relative 2 = entry-sequenced 3 = key-sequenced 4 = edit 8 = odd unstructured .⟨4:9⟩ = device type .⟨10:15⟩ = device subtype The device type and subtype are described in Appendix B. File types 0-3 are described in the *ENSCRIBE Programming Manual.*
FILE^FILENAME^ADDR		⟨filename⟩ returns the word address within the FCB of the physical file name.
FILE^FNUM	Open	⟨filenum⟩ returns the file number. If the file is not open, the file number is −1.
FILE^FNUM^ADDR		⟨filenum⟩ returns the word address within the FCB of the file number.
FILE^FWDLINKFCB		⟨forward-link-pointer⟩ returns the address of the FCB pointed to by the forward link pointer within the FCB. This value indicates the linked-to FCBs that need to be checkpointed after an OPEN^FILE or CLOSE^FILE.
FILE^LOGICALFILENAME^ADDR		@⟨logical file name⟩ returns the word address within the FCB of the logical file name. The logical file name is encoded as follows: byte numbers [0] [1] [8] ⟨len⟩⟨logical file name⟩ ⟨len⟩ is the length of the logical file name in bytes {0:7}.

Table 3-1. CHECK^FILE Operations (Continued)

⟨operation⟩	State of File	⟨retval⟩
FILE∧LOGIOOUT	Open	⟨state of the logioout bit⟩ 0 to indicate there is no logical i/O outstanding. 1 if a logical READ is outstanding. 2 if a logical WRITE is outstanding.
FILE∧OPENACCESS		⟨open access⟩ returns the open access for the file. See SET∧FILE for the format.
FILE∧OPENEXCLUSION	Open	⟨exclusion⟩ returns the open exclusion for the file. See SET∧FILE for the format.
FILE∧PHYSIOOUT	Open	⟨state of the physioout bit⟩ 0 to indicate there is no outstanding physical I/O operation. 1 if a physical I/O operation is outstanding.
FILE∧PRIEXT		⟨primary extent size⟩ returns the file's primary extent size in pages.
FILE∧PRINT∧ERR∧MSG	Open	⟨state of print errmsg bit⟩ 0 if no error message is to be printed upon detection of a fatal error in the file. 1 if an error message is to be printed.
FILE∧PROMPT	Open	⟨interactive prompt character⟩ returns the interactive prompt character for the file in ⟨9:15⟩.
FILE∧RCVEOF	Open	⟨state of rcveof bit⟩ 0 if the user does not get an end-of-file (EOF) indication when the process [pair] having this process open closes it. 1 if the user does get an EOF indication when this process closes.
FILE∧RCVOPENCNT	Open	⟨$RECEIVE opener count⟩ returns a count of current openers for this process {0:2}. At any given moment, openers are limited to a single process [pair].
FILE∧RCVUSEROPENREPLY	Open	⟨state of the rcv-user-open-reply bit⟩ 0 if the SIO procedures are to reply to the OPEN messages ($RECEIVE file). 1 if the user is to reply to the OPEN messages.
FILE∧READ∧TRIM	Open	⟨state of the read trim bit⟩ 0 if the trailing blanks are not trimmed off the data read from this file. 1 if the trailing blanks are trimmed.
FILE∧RECORDLEN		⟨record length⟩ returns the logical record length.

Table 3-1. CHECK^FILE Operations (Continued)

⟨operation⟩	State of File	⟨retval⟩
FILE^SECEXT		⟨secondary extent size⟩ returns the file's secondary extent size in pages.
FILE^SEQNUM^ADDR		@⟨sequence number⟩ returns the word address within the FCB of an INT (32) sequence number. This is the line number of the last record of an edit file. For a nonedit file, this is the sequence number of the last record multiplied by 1000.
FILE^SYSTEMMESSAGES	Open	⟨system message mask⟩ returns a mask word indicating which system messages the user handles directly. See SET^FILE for the format. A 0 indicates that the sequential I/O procedures handle all system messages. Note that this operation cannot check some of the newer system messages; for these, use FILE^SYSTEMMESSAGESMANY.
FILE^SYSTEMMESSAGESMANY	Open	@⟨system message mask words⟩ returns a four-word mask indicating which system messages the user handles directly. See SET^FILE for the format. A return of all zeros indicates that the SIO procedures handle all system messages.
FILE^TRACEBACK		⟨state of traceback bit⟩ 0 if the P-relative address should not be appended to all SIO error messages. 1 if the P-relative address should be appended to all SIO error messages.
FILE^USERFLAG		⟨user flag⟩ returns the user flag word. (See SET^FLAG procedure, SET^USERFLAG operation.)
FILE^USERFLAG^ADDR		@⟨user flag⟩ returns the word address within the FCB of the user flag word.
FILE^WRITE^FOLD	Open	⟨state of the write-fold bit⟩ 0 if records longer than the logical record length are truncated. 1 if long records are folded.
FILE^WRITE^PAD	Open	⟨state of write-pad bit⟩ 0 if a record shorter than the logical record length is not padded with trailing blanks before it is written to the file. 1 if a short record is padded with trailing blanks.
FILE^WRITE^TRIM	Open	⟨state of the write-trim bit⟩ 0 if trailing blanks are not trimmed from data written to the file. 1 if trailing blanks are trimmed.

Example

```
@INFILE^NAME := CHECK^FILE ( IN^FILE , FILE^FILENAME^ADDR );
```

Related Programming Manual

For programming information about the CHECK^FILE procedure, refer to the GUARDIAN Operating System Programmer's Guide.

CLOSE^FILE PROCEDURE

The CLOSE^FILE procedure is used to close a file.

The syntax for CLOSE^FILE is:

```
{ <error> := } CLOSE^FILE ( { <common-fcb> }          ! i
{ CALL       }              { <file-fcb>   }          ! i

  ,[ <tape-disposition> ] );                          ! i
```

<error> — returned value

INT

returns either a file system or a sequential I/O (SIO) procedure error number indicating the outcome of the close. In any case, the file is closed.

If the abort-on-error mode (the default) is in effect, the only possible value for <error> is 0.

<common-fcb> — input

INT:ref:*

identifies the file to be closed (if the file control block (FCB) is passed) or indicates that all open files are closed (if the common FCB is passed). If BREAK is owned for any file being closed, it is returned to its previous owner.

<file-fcb> — input

INT:ref:*

identifies the file to be closed if the file control block (FCB) is passed. If BREAK is owned for any file being closed, it is returned to its previous owner.

→

```
<tape-disposition>                input

  INT:value

  specifies mag tape disposition.

    <tape-disposition>.<13:15> denotes:

      0 = rewind, unload, don't wait for completion.
      1 = rewind, take offline, don't wait for completion.
      2 = rewind, leave online, don't wait for completion.
      3 = rewind, leave online, wait for completion.
      4 = do not rewind, leave online.
```

Condition Code Settings

The condition code has no meaning following a call to CLOSE^FILE.

Considerations

- When to Use CLOSE^FILE

  Data can be lost if a WRITE^FILE with a count of -1 is not specified or a CLOSE^FILE is not performed against EDIT files or files that are opened with WRITE access and blocking capability before the process is deleted.

- If BREAK is taken, CLOSE^FILE gives BREAK (if owned) to its previous owner.

- For tapes with WRITE access, SIO writes two end-of-file marks, (control 2).

- CLOSE^FILE completes all outstanding nowait I/O operations on files that are to be closed.

- $RECEIVE and CLOSE^FILE

  If the file is $RECEIVE and the user is not handling close messages, SIO waits for a message from each opener. It then replies with either error 45, if READ-only access, or error 1, if READ/WRITE access, until there are no more openers (each opener has closed the process by calling CLOSE^FILE).

Example

```
CALL CLOSE^FILE ( COMMON^FCB );          ! closes all files.
```

Related Programming Manual

For programming information about the CLOSE^FILE procedure, refer to the GUARDIAN Operating System Programmer's Guide.

GIVE^BREAK PROCEDURE

The GIVE^BREAK procedure returns BREAK to the previous owner (the process that owned BREAK before the last call to TAKE^BREAK).

The syntax for GIVE^BREAK is:

```
{ <error> := } GIVE^BREAK ( { <common-fcb> } );        ! i
{ CALL       }              { <file-fcb>   }           ! i
```

<error> returned value

INT

returns a file system or sequential I/O procedure error indicating the outcome of the operation.

<common-fcb> input

INT:ref:*

identifies the file returning BREAK to the previous owner. <common-fcb> is allowed for convenience. If BREAK is not owned, this call is ignored.

<file-fcb> input

INT:ref:*

identifies the file returning BREAK to the previous owner. If BREAK is not owned, this call is ignored.

Condition Code Settings

The condition code has no meaning following a call to GIVE^BREAK.

Example

```
CALL GIVE^BREAK ( OUT^FILE );  ! return BREAK to previous owner.
```

Related Programming Manual

For programming information about the GIVE^BREAK procedure, refer to the GUARDIAN Operating System Programmer's Guide.

NO^ERROR PROCEDURE

NO^ERROR is called internally by sequential I/O (SIO) procedures. Error handling and retries are implemented within the SIO procedure environment by the NO^ERROR procedure.

If the file is opened by OPEN^FILE, then the NO^ERROR procedure can be called directly for the file system procedures.

The syntax for NO^ERROR is:

```
{ <no-retry> := } NO^ERROR   <state>                    ! i
{ CALL          }           ,<file-fcb>                 ! i
                            ,<good-error-list>          ! i
                            ,<retryable>  );            ! i
```

<no-retry> returned value

INT

indicates whether or not the I/O operation should be retried. Values of <no-retry> are:

```
    0 = operation should be retried.
  <>0 = operation should not be retried.
```

If <no-retry> is not 0, one of the following is indicated:

- <state> is not 0.
- No error occurred; error is 0.
- error is a good error number on the list.
- Fatal error occurred, and abort-on-error mode is off.
- Error is a BREAK error, and BREAK is enabled for <file-fcb>.

<state> input

INT:value

if nonzero, indicates the operation is considered successful. The file error and retry count variables in the file control block (FCB) are set to zero, with <no-retry> returned as nonzero. Typically, either of two values is passed in this position:

⟶

= (CCE) immediately follows a file system call. If equal is true, the operation is successful. This eliminates a call to FILEINFO by NO^ERROR.

0 forces NO^ERROR to first check the error value in the FCB. If the FCB error is 0, NO^ERROR calls FILEINFO for the file.

<file-fcb> input

INT:ref:*

identifies the file to be checked.

<good-error-list> input

INT:ref:*

is a list of error numbers; if one of the numbers matches the current error, <no-retry> is returned as nonzero (no retry). The format of <good-error-list>, in words, is:

```
word [ 0 ] = number of error numbers in list {0:n}
word [ 1 ] = good error number
       .
       .
       .
word [ n ] = good error number.
```

<retryable> input

INT:value

is used to determine whether certain path errors should be retried. If <retryable> is not zero, errors in the range of {120, 190, 202:231} cause retry according to the device type as follows:

Device	Retry Indication
Operator	Yes
Process	NA
$RECEIVE	NA
Disc	(opened with sync depth of 1, so not applicable)

→

Terminal	Yes
Printer	Yes
Mag Tape	No
Card Reader	No

If the path error is either of {200:201}, a retry indication is given in all cases following the first attempt.

Condition Code Settings

The condition code has no meaning following a call to NO^ERROR.

Example

```
INT GOOD^ERROR [ 0:1 ] := [ 1, 11 ]; ! nonexistent record.
      .
      .
      .
NO^ERROR ( = , OUT^FILE , GOOD^ERROR , FALSE );
```

Related Programming Manual

For programming information about the NO^ERROR procedure, refer to the GUARDIAN Operating System Programmer's Guide.

OPEN^FILE PROCEDURE

The OPEN^FILE procedure permits access to a file when using sequential I/O (SIO) procedures.

The syntax for OPEN^FILE is:

```
{ <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
```

<error> returned value

INT

returns a file system or SIO procedure error number indicating the outcome of the operation.

If the abort-on-open-error mode (the default) is in effect, the only possible value of <error> is 0.

<common-fcb> input

INT:ref:*

is an array of FCBSIZE words for use by the SIO procedures. Only one common file control block (FCB) is used per process. This means the same data block is passed to all OPEN^FILE calls. The first word of the common FCB must be initialized to 0 before the first OPEN^FILE call following a process startup.

→

<file-fcb> input

INT:ref:*

is an array of FCBSIZE words for use by the SIO procedures. The file FCB uniquely identifies this file to other SIO procedures. The file FCB must be initialized with the name of the file to be opened before the OPEN^FILE call is made.

Refer to the GUARDIAN Operating System Programmer's Guide for information about the FCB structure.

<block-buffer> input

INT:ref:*

is an array used for one of three different purposes:

1. When reading a structured file, the buffer (if large enough) is used by GUARDIAN for ENSCRIBE sequential block buffering.

2. When reading or writing an EDIT file, the buffer is used by SIO to contain part of the EDIT file directory and EDIT file pages being assembled or disassembled. The buffer must be supplied for an EDIT file.

3. If the <block-buffer> is not being used for either 1 or 2, then the array is used for SIO record blocking and deblocking. No blocking is performed if any of the following occurs:

   - <block-buffer> or <block-bufferlen> is omitted.
   - The <block-bufferlen> is insufficient according to the record length for the file.
   - READ/WRITE access is indicated.

   Blocking occurs when this parameter is supplied, the block buffer is of sufficient length (as indicated by the <block-bufferlen> parameter), and blocking is appropriate for the device.

The block buffer must be located within 'G'[0:32767] of the data area.

→

<block-bufferlen> input

INT:value

indicates the length, in bytes, of the block buffer. This length must be able to contain at least one logical record. For an EDIT file, the minimum length on READ is 144 bytes; on WRITE, the minimum length is 1024 bytes.

<flags> input

INT(32):value

if present, is used in conjunction with the <flags-mask> parameter to set file transfer characteristics. If omitted, all positions are treated as zeros. The following literals can be combined using signed addition since bit 0 is not used:

ABORT^OPENERR	CRLF^BREAK	READ^TRIM
ABORT^XFERERR	MUSTBENEW	VAR^FORMAT
AUTO^CREATE	NOWAIT	WRITE^FOLD
AUTO^TOF	PRINT^ERR^MSG	WRITE^PAD
BLOCKED	PURGE^DATA	WRITE^TRIM

(See "Considerations" for the meanings of literals used with <flags>.)

<flags-mask> input

INT(32):value

specifies which bits of the flag field are used to alter the file transfer characteristics. The characteristic to be altered is indicated by entering a 1 in the bit position corresponding to the <flags> parameter. A 0 indicates the default setting is used. When omitted, all positions are treated as zeros.

<max-recordlen> input

INT:value

specifies the maximum record length for records within this file. If omitted, the maximum record length is 132.

→

The open is aborted with an SIOERR^INVALIDRECLENGTH, error 520, if the file's record length exceeds the maximum record length, and <max-recordlen> is not 0. If <max-recordlen> is 0, then any record length is permitted.

<prompt-char> input

INT:value

is used to set the interactive prompt character for reading from terminals or processes. When not supplied, the prompt defaults to "?". The prompt character is limited to seven bits, <9:15>.

<error-file-fcb> input

INT:ref:*

if present, specifies a file where error messages are displayed for all files. Only one error reporting file is allowed per process. The file specified in the latest OPEN^FILE is the one used. Omitting this parameter does not alter the setting of the current error reporting file.

The default error reporting file is the home terminal.

If the error reporting file is not open when needed, it is opened only for the duration of the message printing, then closed. Remember that the error file FCB must be initialized. Refer to the GUARDIAN Operating System Programmer's Guide for information about the file FCB.

Condition Code Settings

The condition code has no meaning following a call to OPEN^FILE.

Considerations

- Specifics of AUTO^TOF

  If AUTO^TOF is on, a top-of-form control operation is performed to the file when (1) the file being opened is a process or a line printer and (2) WRITE or READ/WRITE access is specified.

- When READ/WRITE Access Is not Permitted

  If the file is an EDIT file or if blocking is specified, either READ or WRITE access must be specified for the OPEN to succeed. READ/WRITE access is not permitted.

- Accessing a Temporary Disc File

  When using OPEN^FILE to access a temporary disc file, AUTO^CREATE must be disabled; otherwise, the OPEN^FILE call results in a file system error 13.

- Sync Depth of Open Files

  All files opened with the OPEN^FILE procedure are opened with a sync depth of one. One is the only possible sync depth; no other can be set.

- Error Reporting File

  The error reporting file is used, when possible, for reporting errors. If this file cannot be used or the error is with the error reporting file, the default error reporting file is used.

- SIO procedures append data to the file if access is WRITE only, and PURGE^DATA is off (default).

- List of Literals Used With <flags> and <flags-mask>

  ABORT^OPENERR — abort on open error, defaults to on. If on, and a fatal error occurs during the OPEN^FILE, all files are closed, and the process abnormally ends. If off, the file system or SIO procedure error number returns to the caller.

  ABORT^XFERERR — abort on data transfer error, defaults to on. If on, and a fatal error occurs during a data transfer operation (such as a call to any SIO procedure except OPEN^FILE), all files are closed and the process abnormally ends. If off, the file system or the SIO procedure error number returns to the caller.

  AUTO^CREATE — auto create, defaults to on. If on, and open access is WRITE, a file is created, if one is not already there. If WRITE access is not given and the file does not exist, error 11 is returned. If no file code has been assigned, or if the file code is 101, and a block buffer of at least 1024 bytes is provided, an EDIT file is created. If there is not a buffer of sufficient size and no new file code is specified, then a file code of 0 is used.

The default extent sizes are 4 pages for the primary extent and 16 pages for the secondary extent.

AUTO^TOF	auto top of form, defaults to on. If on, and the file is open with WRITE access and is a line printer or process, a page eject is issued to the file within the OPEN^FILE procedure.
BLOCKED	nondisc blocking, defaults to off. A block buffer of sufficient length must also be specified.
CRLF^BREAK	carriage return/line feed (CR/LF) on BREAK, defaults to on. If on and BREAK is enabled, a CR/LF is written to the terminal when BREAK is typed.
MUSTBENEW	file must be new, defaults to off. This applies only if AUTO^CREATE is specified. If the file already exists, error 10 returns.
NOWAIT	nowait I/O, defaults to off (wait I/O). If on, nowait I/O is in effect. If NOWAIT is specified in the open flags of OPEN^FILE, then the nowait depth is 1. It is not possible to use a nowait depth of greater than 1 using SIO procedures.
PRINT^ERR^MSG	print error message, defaults to on. If on, and a fatal error occurs, an error message is displayed on the error file. This is the home terminal unless otherwise specified.
PURGE^DATA	purge data, defaults to off. If on, and open access is WRITE, the data is purged from the file after the OPEN. If off, the data is appended to the existing data.
READ^TRIM	read trailing blank trim, defaults to on. If on, the <count-read> parameter does not account for trailing blanks.
VAR^FORMAT	variable-length records, defaults to off for fixed-length records. If on, the maximum record length for variable-length records is 254 bytes.
WRITE^FOLD	write fold, defaults to on. If on, WRITEs that exceed the record length cause multiple logical records to be written. If off, WRITEs that exceed the record length are truncated to record-length bytes; no error message or warning is given.

WRITE^PAD	write blank pad, defaults to on for disc fixed-length records and off for all other files. If on, WRITEs of less than record-length bytes, including the last record if WRITE^FOLD is in effect, are padded with trailing blanks to fill out the logical record.
WRITE^TRIM	write trailing blank trim, defaults to on. If on, trailing blanks are trimmed from the output record before being written to the file.

Example

```
ERROR := OPEN^FILE ( COMMON^FCB , IN^FILE , BUFFER , BUFFER^SIZE
                   , FLAGS , FLAGS^MASK , PROMPT );
```

Related Programming Manual

For programming information about the OPEN^FILE procedure, refer to the GUARDIAN Operating System Programmer's Guide.

READ^FILE PROCEDURE

The READ^FILE procedure is used to read a file sequentially.

The file must be able to be opened with READ or READ/WRITE access with file system procedure calls, which is equivalent to READ or WRITE access under the sequential I/O (SIO) procedures.

The syntax for READ^FILE is:

```
{ <error> := } READ^FILE ( <file-fcb>                  ! i
{ CALL       }             ,<buffer>                   ! o
                           ,[ <count-read> ]           ! o
                           ,[ <prompt-count> ]         ! i
                           ,[ <max-read-count> ]       ! i
                           ,[ <no-wait> ]  );          ! i
```

<error> returned value

INT

returns a file system or SIO procedure error indicating the outcome of the READ.

If abort-on-error mode is in effect, the only possible values for <error> are:

```
   0 = no error
   1 = end of file
   6 = system message (only if user requested system messages
       through SET^SYSTEMMESSAGES or SET^SYSTEMMESSAGESMANY)
 111 = operation aborted because of BREAK (if BREAK is enabled)
```

If <no wait> is not zero, and if abort-on-error is in effect, the only possible value for <error> is 0.

<file-fcb> input

INT:ref:*

identifies the file to be read.

→

<buffer> output

INT:ref:*

is where the data is returned. The buffer must be located within 'G'[0:32767] process data area.

<count-read> output

INT:ref:1

returns the number of bytes returned to <buffer>. If <no-wait> is not zero, then this parameter has no meaning and can be omitted. The count is then obtained in the call to WAIT^FILE.

<prompt-count> input

INT:value

is a count of the number of bytes in <buffer>, starting with element zero, to be used as an interactive prompt for terminals or interprocess files. If omitted, the interactive prompt character defined in OPEN^FILE is used.

<max-read-count> input

INT:value

specifies the maximum number of bytes to be returned to <buffer>. If omitted or if it exceeds the file's logical record length, the logical record length is used for this file.

<no-wait> input

INT:value

indicates whether or not to wait for the I/O operation to complete in this call. If omitted or zero, then "wait" is indicated. If not zero, the I/O operation must be completed in a call to WAIT^FILE.

Condition Code Settings

The condition code has no meaning following a call to READ^FILE.

Considerations

- Terminal or Process File

  If the file is a terminal or process, a WRITEREAD operation is performed using the interactive prompt character or <prompt-count> character from <buffer>. For $RECEIVE, READ^FILE does a READUPDATE instead of a READ.

Example

```
ERROR := READ^FILE ( IN^FILE , BUFFER , COUNT );
```

Related Programming Manual

For programming information about the READ^FILE procedure, refer to the GUARDIAN Operating System Programmer's Guide.

SET^FILE PROCEDURE

The SET^FILE procedure alters file characteristics and checks the old value of those characteristics being altered.

The syntax for SET^FILE is:

```
{ <error> := } SET^FILE ( { <common-fcb> }                    ! i
{ CALL       }            { <file-fcb>   }                    ! i

  , <operation>                                               ! i
  , [ <new-value> ]                                           ! i
  , [ <old-value> ]  );                                       ! o
```

<error> returned value

INT

returns a file system or sequential I/O (SIO) procedure error number indicating the outcome of the SET^FILE.

If abort-on-error mode is in effect, the only possible value for <error> is 0.

<common-fcb> input

INT:ref:*

identifies those files whose characteristics are to be altered.

The SET^FILE operations that make sense only for the <common-fcb> are the SET^BREAKHIT, SET^ERRORFILE, and SET^TRACEBACK.

When using INIT^FILEFCB, FILE^FWDLINKFCB, FILE^BWDLINKFCB, and SET^TRACEBACK the FCB can be specified as the <common-fcb> or the <file-fcb>.

If an improper FCB is specified or the FCB is not initialized, an error is indicated.

→

<file-fcb> input

INT:ref:*

identifies the file whose characteristics are to be altered. In most cases, the FCB must be associated with a file or $RECEIVE (in this case you would specify the <file-fcb>).

When using INIT^FILEFCB, FILE^FWDLINKFCB, FILE^BWDLINKFCB, and SET^TRACEBACK the FCB can be specified as the <common-fcb> or the <file-fcb>.

If an improper FCB is specified or the FCB is not initialized, an error is indicated.

<operation> input

INT:value

specifies the file characteristic to be altered. (See Table 3-2.)

<new-value> input

INT:value

specifies a new value for the specified <operation>. This is optional, depending on the operation desired.

<old-value> output

INT:ref:*

is a variable in which the current value for the specified <operation> returns. This can vary from 1 to 12 words and is useful in saving this value for reset later. If <old-value> is omitted, the current value is not returned.

Condition Code Settings

The condition code has no meaning following a call to SET^FILE.

Considerations

In Table 3-2, the column labeled "State of File" is flagged with the following:

open = the file must be opened to alter the file's characteristics.

closed = the file must be closed to alter the file's characteristics.

blank = the column is left blank, indicating the file can be either open or closed.

Table 3-2. SET^FILE Operations

⟨operation⟩ Parameter (Required)	Description of Operation Requested	⟨new-value⟩ Parameter Entered (Optional)	⟨old-value⟩ Parameter Entered (Optional)	State of File
ASSIGN∧BLOCKBUFLEN (or ASSIGN∧BLOCKLENGTH)	Specifies the block length, in bytes, for the file.	⟨new-blocklen⟩	⟨blocklen⟩	Closed
ASSIGN∧FILECODE	Specifies the file code for the file.	⟨new-file-code⟩	⟨file-code⟩	Closed
ASSIGN∧FILENAME	Specifies the physical name of the file to be opened. This operation is not used when the INITIALIZER procedure is called to initialize the FCBs. For example: CALL SET∧FILE (in∧file ,ASSIGN∧FILENAME ,@in∧filename);	@⟨filename⟩	⟨filename⟩ FOR 12-words	Closed
ASSIGN∧LOGICALFILENAME	Specifies the logical name of the file to be opened. The ⟨logical-filename⟩ must be encoded as follows: byte numbers [0] [1] [8] ⟨len⟩⟨logical-filename⟩ ⟨len⟩ is the length of the logical filename ⟨0:7⟩.	@⟨logical-filename⟩	@⟨logical-filename⟩ FOR 4-words	Closed
ASSIGN∧OPENACCESS	Specifies the open access for the file. The following literals are provided for ⟨open-access⟩: READWRITE∧ACCESS (0) READ∧ACCESS (1) WRITE∧ACCESS (2) Even if READ∧ACCESS is specified, SIO actually opens the file with READWRITE∧ACCESS to facilitate interactive I/O.	⟨new-open-access⟩	⟨open-access⟩	Closed
ASSIGN∧OPENEXCLUSION	Specifies the open exclusion for the file. The following literals are provided for ⟨open-exclusion⟩: SHARED (0) EXCLUSIVE (1) PROTECTED (3)	⟨new-open-exclusion⟩	⟨open-exclusion⟩	Closed
ASSIGN∧PRIEXT (or ASSIGN∧PRIMARYEXTENTSIZE)	Specifies the primary extent size (in units of 2048-byte blocks) for the file.	⟨new-pre-ext-size⟩	⟨pri-ext-size⟩	Closed
ASSIGN∧RECORDLEN (or ASSIGN∧RECORDLENGTH)	Specifies the logical record length (in bytes) for the file. ASSIGN∧RECORDLENGTH gives the default READ or WRITE count. For defaults, refer to the *GUARDIAN Operating System Programmer's Guide.*	⟨new-recordlen⟩	⟨recordlen⟩	Closed
ASSIGN∧SECEXT (or ASSIGN∧SECONDARYEXTENTSIZE)	Specifies the secondary extent size (in units of 2048-byte blocks) for the file.	⟨new-sec-ext-size⟩	⟨sec-ext-size⟩	Closed

Table 3-2. SET^FILE Operations (Continued)

⟨operation⟩ Parameter (Required)	Description of Operation Requested	⟨new-value⟩ Parameter Entered (Optional)	⟨old-value⟩ Parameter Entered (Optional)	State of File
INIT^FILEFCB	Specifies that the file FCB be initialized. This operation is not used when the !NITIALIZER procedure is called to initialize the FCBs. For example: CALL SET^FILE (common^fcb ,INIT^FILEFCB); CALL SET^FILE (in^file ,INIT^FILEFCB);	must be omitted	must be omitted	Closed
SET^ABORT^XFERERR	Sets or clears abort-on-transfer error for the file. If on, and a fatal error occurs during a data transfer operation (such as a call to any SIO procedure except OPEN^FILE), all files are closed and the process abnormally ends. If off, the file system or SIO procedure error number returns to the caller.	⟨new-state⟩	⟨state⟩	Open
SET^BREAKHIT	Sets or clears break hit for the file. This is used only if the user is handling BREAK independently of the SIO procedures, or if the user has requested BREAK system messages through SET^SYSTEMMESSAGES or SET^SYSTEMMESSAGESMANY.	⟨new-state⟩	⟨state⟩	
SET^CHECKSUM	Sets or clears the checksum word in the FCB. This is useful after modifying an FCB directly (that is, without using the SIO procedures).	⟨new-checksum-word⟩	⟨checksum-word-in-fcb⟩	
SET^COUNTXFERRED	Sets the physical I/O count, in bytes, transferred for the file. This is used only if nowait I/O is in effect, and the user is making the call to AWAITIO for the file. This is the ⟨count-transferred⟩ parameter value returned from AWAITIO.	⟨new-count⟩	⟨count⟩	Open
SET^CRLF^BREAK	Sets or clears carriage return/line feed (CR/LF) on BREAK for the file. If on, a CR/LF is executed on the terminal when the BREAK key is typed.	⟨new-state⟩	⟨state⟩	Open
SET^DUPFILE	Specifies a duplicate file for the file. This is a file where data read from ⟨file-fcb⟩ is printed. The default is no duplicate file. For example: CALL SET^FILE (in^file, SET^DUPFILE, @out^file);	@⟨new-dup-file-fcb⟩	@⟨dup-file-fcb⟩	Open
SET^EDITREAD^REPOSITION	Specifies that the following READ^FILE is to begin at the position set in the sequential block buffer (second through fourth words). For example: CALL SET^FILE (EDIT^FCB, SET^EDITREAD^REPOSITION);	must be omitted	must be omitted	Open
SET^ERROR	Sets file system error code value for the file. This is used only if nowait I/O is in effect, and the user makes the call to AWAITIO for the file. This is the ⟨error⟩ parameter value returned from FILEINFO.	⟨new-error⟩	⟨error⟩	Open

Table 3-2. SET^FILE Operations (Continued)

⟨operation⟩ Parameter (Required)	Description of Operation Requested	⟨new-value⟩ Parameter Entered (Optional)	⟨old-value⟩ Parameter Entered (Optional)	State of File
SET∧ERRORFILE	Sets error reporting file for all files. Defaults to home terminal. If the error reporting file is not open when needed by the SIO procedures, it is opened for the duration of the message printing, then closed.	@⟨new-error-file-fcb⟩	@⟨error-file-fcb⟩	
SET∧OPENERSPID	Sets the allowable openers ⟨process-id⟩ for $RECEIVE file. This is used to restrict the openers of this process to a specified process. A typical example is using the SIO procedures to read the startup message. **NOTE** If open message = 1 is specified to SET∧SYSTEMMESSAGES or SET∧SYSTEMMESSAGESMANY, the setting of SET∧OPENERSPID has no meaning.	@⟨openers-pid⟩	⟨openers-pid⟩ FOR 4-words	Open
SET∧PHYSIOOUT	Sets or clears physical I/O outstanding for the file specified by ⟨file-fcb⟩. This is used only if nowait I/O is in effect, and the user makes the call to AWAITIO for the file.	⟨new-state⟩	⟨state⟩	Open
SET∧PRINT∧ERR∧MSG	Sets or clears print error message for the file. If on, and a fatal error occurs, an error message is displayed on the error file. This is the home terminal unless otherwise specified.	⟨new-state⟩	⟨state⟩	Open
SET∧PROMPT	Sets interactive prompt for the file. See the OPEN∧FILE procedure.	⟨new-prompt-char⟩	⟨prompt-car⟩	Open
SET∧RCVEOF	Sets return end of file (EOF) on process close for $RECEIVE file. This causes an EOF indication to be returned from READ∧FILE when the receive open count goes from 1 to 0; the last close message is received. The setting for return EOF has no meaning if the user is monitoring OPEN and CLOSE messages. If the file is opened with READ-only access, the setting defaults to on for return EOF.	⟨new-state⟩	⟨state⟩	Open
SET∧RCVOPENCNT	Sets receive open count for the $RECEIVE file. This operation is intended to clear the count of openers when an OPEN already accepted by the SIO procedures is subsequently rejected by the user. See SET∧RCVUSEROPENREPLY.	⟨new-receive-open-count⟩	⟨receive-open-count⟩	Open
SET∧RCVUSEROPENREPLY	Sets user-will-reply for the $RECEIVE file. This is used if the SIO procedures are to maintain the opener's directory, thereby limiting OPENs to a single process or a process pair but keeping the option to reject OPENs.	⟨new-state⟩	⟨state⟩	Open

Table 3-2. SET^FILE Operations (Continued)

⟨operation⟩ Parameter (Required)	Description of Operation Requested	⟨new-value⟩ Parameter Entered (Optional)	⟨old-value⟩ Parameter Entered (Optional)	State of File
	If ⟨state⟩ is 1, an ⟨error⟩ of 6 returns from a call to READ∧FILE when an OPEN message is received and is the only current OPEN by a process or a process pair. If an OPEN is attempted by a process, and an OPEN is currently in effect, the OPEN attempt is rejected by the SIO procedures; no return is made from READ∧FILE because of the rejected OPEN attempt. If ⟨state⟩ is 0, a return from READ∧FILE is made only when data is received. **NOTE** If open message = 1 is specified to SET∧SYSTEMMESSAGES or SET∧SYSTEMMESSAGESMANY, the setting of SET∧RCVUSEROPENREPLY has no meaning. An ⟨error⟩ of 6 returns from READ∧FILE if an open message is accepted by the SIO procedures.			
SET∧READ∧TRIM	Sets or clears read-trailing-blank-trim for the file. If on, the ⟨count-read⟩ parameter does not account for trailing blanks.	⟨new-state⟩	⟨state⟩	Open
SET∧SYSTEMMESSAGES	Sets system message reception for the $RECEIVE file. Setting a bit in the ⟨sys-msg-mask⟩ indicates that the corresponding message is to pass back to the user. Default action is for the SIO procedures to handle all system messages. ⟨sys-msg-mask⟩[0] .⟨0⟩ = BREAK message .⟨1⟩ = unused .⟨2⟩ = CPU Down message .⟨3⟩ = CPU Up message .⟨4⟩ = unused .⟨5⟩ = STOP message .⟨6⟩ = ABEND message .⟨7⟩ = unused .⟨8⟩ = MONITORNET message .⟨9⟩ = unused .⟨10⟩ = OPEN message .⟨11⟩ = CLOSE message .⟨12⟩ = CONTROL message .⟨13⟩ = SETMODE message .⟨14⟩ = RESETSYNC message .⟨15⟩ = unused The user replies to the system messages designated by this operation by using WRITE∧FILE. If no WRITE∧FILE is encountered before the next READ∧FILE, a ⟨reply-error-code⟩ = 0 is made automatically. Note that this operation cannot set some of the newer system messages; for these, use SET∧SYSTEMMESSAGESMANY.	⟨new-sys-msg-mask⟩	⟨sys-sys-mask⟩	Open

Table 3-2. SET^FILE Operations (Continued)

⟨operation⟩ Parameter (Required)	Description of Operation Requested	⟨new-value⟩ Parameter Entered (Optional)	⟨old-value⟩ Parameter Entered (Optional)	State of File
SET^SYSTEMMESSAGESMANY	Sets system message reception for the $RECEIVE file. ⟨sys-msg-mask-words⟩ is a four-word mask. Setting a bit in the ⟨sys-msg-mask-words⟩ indicates that the corresponding message is to pass back to the user. Default action is for the SIO procedures to handle all system messages. ⟨sys-msg-mask⟩[0] .⟨0:1⟩ = unused .⟨2⟩ = CPU Down message .⟨3⟩ = CPU Up message .⟨4⟩ = unused .⟨5⟩ = STOP message .⟨6⟩ = ABEND message .⟨7⟩ = unused .⟨8⟩ = MONITORNET message .⟨9⟩ = unused .⟨10⟩ = SETTIME message (NonStop II systems only) .⟨11⟩ = Power On message (NonStop II systems only) .⟨12⟩ = NEWPROCESSNOWAIT message (NonStop II systems only) .⟨13:15⟩ = unused ⟨sys-msg-mask⟩[1] .⟨0:3⟩ = unused .⟨4⟩ = BREAK message .⟨5⟩ = unused .⟨6⟩ = Time Signal message (NonStop II systems only) .⟨7⟩ = Memory Lock Completion message (NonStop II systems only) .⟨8⟩ = Memory Lock Failure message (NonStop II systems only) .⟨9:13⟩ = unused .⟨14⟩ = OPEN message .⟨15⟩ = CLOSE message ⟨sys-msg-mask⟩[2] .⟨0⟩ = CONTROL message .⟨1⟩ = SETMODE message .⟨2⟩ = RESETSYNC message .⟨3⟩ = CONTROLBUF message .⟨4:15⟩ = unused ⟨sys-msg-mask⟩[3] .⟨0:15⟩ = all bits unused	@⟨new-sys-msg-mask-words⟩	⟨sys-msg-mask-words⟩	Open

Table 3-2. SET^FILE Operations (Continued)

⟨operation⟩ Parameter (Required)	Description of Operation Requested	⟨new-value⟩ Parameter Entered (Optional)	⟨old-value⟩ Parameter Entered (Optional)	State of File
SET^TRACEBACK	Sets or clears the traceback feature. When traceback is active, the SIO facility appends the caller's P-relative address to all error messages.	⟨new-state⟩	⟨old-state⟩	
SET^USERFLAG	Sets user flag for the file. The user flag is a one-word value in the FCB that the user can manipulate to maintain information about the file.	⟨new-user-flag⟩	⟨user-flag-in-fcb⟩	
SET^WRITE^FOLD	Sets or clears write-fold for the file. If on, WRITEs exceeding the record length cause multiple logical records to be written. If off, WRITEs exceeding the record length are truncated to record-length bytes; no error message or warning is given.	⟨new-state⟩	⟨state⟩	Open
SET^WRITE^PAD	Sets or clears write-blank-pad for the file. If on, WRITEs of less than record-length bytes, including the last record if WRITE^FOLD is in effect, are padded with trailing blanks to fill out the logical record.	⟨new-state⟩	⟨state⟩	Open
SET^WRITE^TRIM	Sets or clears write-trailing-blank-trim for the file. If on, trailing blanks are trimmed from the output record before being written to the file.	⟨new-state⟩	⟨state⟩	Open

Example

```
CALL SET^FILE ( IN^FILE , ASSIGN^FILENAME , @IN^FILENAME );
```

Related Programming Manual

For programming information about the SET^FILE procedure, refer to the GUARDIAN Operating System Programmer's Guide.

TAKE^BREAK PROCEDURE

The TAKE^BREAK procedure enables BREAK monitoring for a file.

The syntax for TAKE^BREAK is:

```
{ <error> := } TAKE^BREAK ( <file-fcb> );          ! i
{ CALL       }
```

<error> output

INT

is a file system or sequential I/O (SIO) procedure error indicating the outcome of the operation.

<file-fcb> input

INT:ref:*

identifies the file for which BREAK is enabled. If the file is not a terminal, or if BREAK is already owned for this file, the call is ignored.

Condition Code Settings

The condition code has no meaning following a call to TAKE^BREAK.

Considerations

- Break Ownership and One Terminal

  Although the GUARDIAN operating system 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"; SIO always issues SETMODE 11 with parameter 2 = 0.

- Taking BREAK Ownership Back

  If a process launches an offspring process that takes BREAK ownership, and the parent process then calls CHECK^BREAK, SIO takes BREAK ownership back. This can affect anticipated handling of BREAK.

Example

```
CALL TAKE^BREAK ( OUT^FILE );
```

Related Programming Manual

For programming information about the TAKE^BREAK procedure, refer to the GUARDIAN Operating System Programmer's Guide.

WAIT^FILE PROCEDURE

The WAIT^FILE procedure is used to wait or check for the completion of an outstanding I/O operation.

The syntax for WAIT^FILE is:

```
<error> := WAIT^FILE ( <file-fcb>                ! i
                     ,[ <count-read> ]           ! o
                     ,[ <time-limit> ] );        ! i
```

<error> returned value

INT

If abort-on-error mode is in effect, the only possible values for <error> are:

- 0 = no error
- 1 = end of file
- 6 = system message (only if user requested system messages through SET^SYSTEMMESSAGES or SET^SYSTEMMESSAGESMANY)
- 40 = operation timed out (only if <time limit> value is supplied and is not -1D)
- 111 = operation aborted because of BREAK (if BREAK is enabled)
- 532 = operation restarted

<file-fcb> input

INT:ref:*

identifies the file for which there is an outstanding I/O operation.

<count-read> output

INT:ref*

if present, is the count of the number of bytes returned due to the requested READ operation. The value returned to the parameter has no meaning when waiting for a WRITE operation to complete.

→

<time-limit>		input
INT(32):value		
if present, indicates whether the caller waits for completion or checks for completion. If omitted, the time limit is set to -1D.		
<time-limit>	<> 0D	indicates a wait for completion. The time limit then specifies the maximum time, in .01-second units, the caller waits for a completion.
	= 0D	indicates a check for completion. WAIT^FILE immediately returns to the caller regardless of whether there is a completion. If no completion occurs, the I/O operation is still outstanding; an <error> 40 and an "operation timed out" message are returned.
(and <error>	= 0D = 40)	There is no completion. Therefore, READ^FILE or WRITE^FILE cannot be called for the file until the operation completes by WAIT^FILE. One method of determining if the operation completes is by the CHECK^FILE operation "FILE^LOGIOOUT."
	= -1D	indicates a willingness to wait forever.

Condition Code Settings

The condition code has no meaning following a call to WAIT^FILE.

Example

```
ERROR := WAIT^FILE ( IN^FILE , COUNT );
```

Related Programming Manual

For programming information about the WAIT^FILE procedure, refer to the GUARDIAN Operating System Programmer's Guide.

WRITE^FILE PROCEDURE

The WRITE^FILE procedure writes a file sequentially. The file must be open with WRITE or READ/WRITE access.

The syntax for WRITE^FILE is:

```
{ <error> := } WRITE^FILE ( <file-fcb>                    ! i
{ CALL       }              ,<buffer>                     ! i
                            ,<write-count>                ! i
                            ,[ <reply-error-code> ]       ! i
                            ,[ <forms-control-code> ]     ! i
                            ,[ <nowait> ]  );             ! i
```

<error> returned value

INT

is a file system or sequential I/O (SIO) error indicating the outcome of the WRITE.

If abort-on-error mode, the default case, is in effect, the only possible values for <error> are:

```
    0 = no error
  111 = operation aborted because of BREAK (if BREAK is
        enabled)
```

If <nowait> is not 0, the only possible value for <error> is 0, when abort-on-error mode is in effect.

<file-fcb> input

INT:ref:*

identifies the file to which data is written.

<buffer> input

INT:ref:*

is the data to be written. <buffer> must be located within 'G'[0:32767], the process data area.

⟶

<write-count> input

INT:value

is the count of the number of bytes of <buffer> to be written. A <write-count> value of -1 causes SIO to flush the block buffer associated with the <file-fcb> passed. For EDIT files, flushing the buffer also updates the EDIT directory on disc.

<reply-error-code> input

INT:value

(for $RECEIVE file only) if present, is a file system error to return to the requesting process by REPLY. If omitted, 0 is returned.

<forms-control-code> input

INT:value

(optional) indicates a forms control operation to be performed prior to executing the actual WRITE when the file is a process or a line printer. <forms-control> corresponds to <parameter> of the file system CONTROL procedure for <operation> equal to 1. No forms control is performed if <forms-control> is omitted, if it is -1, or if the file is not a process or a line printer.

<no-wait> input

INT:value

if present, indicates whether to wait in this call for the I/O to complete. If omitted or zero, then wait is indicated. If <no-wait> is not zero, the I/O must be completed in a call to WAIT^FILE.

Condition Code Settings

The condition code has no meaning following a call to WRITE^FILE.

Example

```
CALL WRITE^FILE ( OUT^FILE ,BUFFER ,COUNT );
```

Related Programming Manual

For programming information about the WRITE^FILE procedure, refer to the GUARDIAN Operating System Programmer's Guide.

APPENDIX A

CONTROL OPERATIONS

This table gives a list of CONTROL operations, that is, those operations used with the I/O devices discussed in this manual for NonStop systems.

CONTROL Operations Table

<operation>	Description	Subtype	<param> Description
1	Terminal or Line Printer Forms Control	0, 2, or 3	0 = form feed (send %014) 1-15 = vertical tab (send %013) 16-79 = skip <param> – 16 lines
	Line Printer	6 or 32	0 = form feed (send %014) 1-15 = single space (send %6412) 16-79 = skip <param> – 16 lines
	Line Printer	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, and so forth) 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 <param> – 16 lines

CONTROL Operations Table (Continued)

<operation>	Description	Subtype	<param> Description
	Line Printer (default DAVFU)	4	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, and so forth) 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 <param> -- 16 lines
2	WRITE end of file on unstructured disc or magnetic tape (if disc, WRITE access is required).	—	None
3	Magnetic tape, rewind and unload, do not wait for completion.	—	None
4	Magnetic tape, take off line, do not wait for completion (treated as <operation> 3 for 5106 Tri-Density Tape Drive).	—	None
5	Magnetic tape, rewind, leave on line, do not wait for completion.	—	None
6	Magnetic tape, rewind, leave on line, wait for completion.	—	None
7	Magnetic tape, space forward files.	—	number of files {0:255}
8	Magnetic tape, space backward files.	—	number of files {0:255}
9	Magnetic tape, space forward records.	—	number of files {0:255}

CONTROL Operations Table (Continued)

<operation>	Description	Subtype	<param> Description
10	Magnetic tape, space backward records.	—	number of files {0:255}
11	Terminal or line printer, wait for modem connect.	3, 4, 6, or 32	None
12	Terminal or line printer, disconnect the modem (that is, hang up).	3, 4, 6, or 32	None
20	Disc, purge data (WRITE access is required).	—	None
21	Disc, allocate or deallocate extents (WRITE access is required).	—	<param> = 0 deallocate all extents past the end-of-file extent
	1:<maximum-extents> = number of extents to allocate for a nonpartitioned file (for DP2 disc files only) 1:16 * number of partitions = number of extents to allocate for a partitioned file		

NOTE

A WRITE end of file (EOF) to an unstructured disc file sets the EOF pointer to the relative byte address indicated by the setting of the next-record pointer and WRITEs the new EOF setting in the file label on disc.

APPENDIX B

DEVICE TYPES AND SUBTYPES

The following table contains a complete list of the device types and subtypes of the NonStop systems.

Device Types and Subtypes Table

Device Type <device-type>.<4:9>	Device Subtype <device-type>.<10:15>
0 = Process	0
1 = Operator Console	0
2 = $RECEIVE	0
3 = Disc (Note: For discs, <device type>.<0> = 1 denotes a removable disc volume; <device type>.<1> = 1 denotes a volume audited by the *Transaction Monitoring Facility (TMF).*	3 = 240 MB capacity (P/N 4104) 4 = 64 MB capacity (P/N 4105, 4106) 5 = 64 MB capacity, movable-head portion (P/N 4109) 6 = 540 MB capacity (P/N 4116) 7 = 1.45 MB capacity, fixed-head portion (P/N 4109) 8 = 128 MB capacity (P/N 4110, 4111) 9 = 264 MB capacity (P/N 4114, 4115)
4 = Magnetic Tape	0 = Nine-Track 1 = Seven-Track (P/N 5105) 2 = Tri-Density Tape Drive (P/N 5106) 3 = 3207 Controller (P/N 5103, 5104)
5 = Line Printer	1 = Drum or Band 3 = Matrix Serial (P/N 5508) 4 = Matrix Serial (P/N 5520) 5 = Band, extended char. set 6 = Letter-quality printer (P/N 5530) 32 = DTR printer
6 = Terminal (conversational or page mode)	0 = Conversational Mode 1 = Page Mode (P/N 6511, 6512) 2 = Page Mode (P/N 6520, 6524) 3 = Page Mode (Remote 6520) 4 = Page Mode (P/N 6530) 5 = Page Mode (Remote 6530)

Device Types and Subtypes Table (Continued)

Device Type <device-type>.<4:9>	Device Subtype <device-type>.<10:15>
6 = Terminal (conversational or page mode) (Continued)	6-10 = Conversational Mode 6 = 3277 (12x40) 7 = 3277 (24x80) 8 = 3277 (32x80) 9 = 3277 (43x80) 10 = 3277 (12x80) 32 = Hard-Copy Console
(SNAX ITI protocol)	20 = 3277 & 3278 model 1 (screen size 12x40) 21 = 3277 & 3278 model 2 (screen size 24x80) 22 = 3278 model 3 (screen size 32x80) 23 = 3278 model 4 (screen size 43x80) 24 = 3278 model 5 (screen size 27x132)
7 = ENVOY Data Communication Line	0 = BISYNC, point-to-point, nonswitched 1 = BISYNC, point-to-point, switched 2 = BISYNC, multipoint, tributary 3 = BISYNC, multipoint, supervisor 8 = ADM-2, multipoint, supervisor 9 = TINET, multipoint, supervisor 10 = Burroughs, multipoint, supervisor 13 = Burroughs, point-to-point, contention 30 = Full-duplex, out line 31 = Full-duplex, in line 40 = Asynchronous line supervisor 50 = Isochronous line 56 = Auto-call unit
8 = Card Reader	0
9 = Process-to-Process Interface	0 = X25AM Process
10 = 3277 CRT Mode Interface	1-5 = Block Mode 1 = 3277 (12x40) 2 = 3277 (24x80) 3 = 3277 (32x80) 4 = 3277 (43x80) 5 = 3277 (12x80) 6 = 328x Printers
(SNAX CRT protocol)	20 = 3277 & 3278 model 1 (screen size 12x40) 21 = 3277 & 3278 model 2 (screen size 24x80) 22 = 3278 model 3 (screen size 32x80) 23 = 3278 model 4 (screen size 43x80) 24 = 3278 model 5 (screen size 27x132) 30 = 328x printer (using CRT protocol)
11 = ENVOYACP Data Communication Line	40 = Synchronous Data Link Control (SDLC) 41 = High-Level Data Link Control (HDLC) 42 = Advanced Data Communication Control Protocol (ADCCP)
12 = TIL (Tandem to IBM Link)	0
13 = SNAX	5 = Service Manager

Device Types and Subtypes Tables (Continued)

Device Type <device-type>.<4:9>	Device Subtype <device-type>.<10:15>
20/23 = TMF	0
26 = THL (Tandem HyperLink)	0
27 = IPB Monitor (FOX)	0
50 = CSM (Communication Subsystem Manager) for 6100 subsystem	0
51 = CP6100	1 = BISYNC 2 = ADCCP 3 = TINET
52 = INFOSAT	0 = Earth station control line 1 = Satellite connect
53 = CSM (ATP6100)	0 = Terminals attached to 6100 subsystem (CLIP1) 1 = CLIP4
58 = SNAX	0 = SDLC line
59 = AM6520	0 = Line to controller 10 = Line to 6100 subsystem
60 = AM3270 TR3271	0 = Line to controller 10 = Line to 6100 subsystem 1 = Line to controller
61 = X.25 Data Communication Line	0-62 = Line to controller (any subtype is accepted) 63 = Line to 6100 subsystem
62 = EXPAND Network Control Process (NCP)	0
63 = EXPAND Line Handler	0 = Single line handler for: —Direct-connect synchronous line —Satellite-connect synchronous line 1 = Multiline path handler for path 2 = Multiline path handler for: —Direct-connect synchronous line —Satellite-connect synchronous line 3 = FOX line handler 5 = Single line handler for direct-connect 6100 subsystem 6 = Multiline path handler for direct-connect 6100 subsystem line

APPENDIX C

SETMODE FUNCTIONS

The following table is a list of SETMODE functions, that are used with the I/O devices discussed in this manual for the NonStop systems.

SETMODE Functions Table

<function>	
1	**Disc:** Set file security <param1> .<0> = 1 for program files only, sets accessor's ID to program file's ID when program file is run (PROGID option). .<1> = 1 sets CLEARONPURGE option on. This means all data in the file is physically erased from the disc (set to zeros) when the file is purged. If this option is not on, the disc space is only logically deallocated on a purge; the data is not destroyed, and another user might be able to examine the "purged" data when the space is reallocated to another file. .<4:6> = ID allowed for reading .<7:9> = ID allowed for writing .<10:12> = ID allowed for execution .<13:15> = ID allowed for purging For each of the fields from .<4:6> through .<13.15>, the value can be any one of the following: 0 = any local ID 1 = member of owner's group (local) 2 = owner (local) 4 = any network user (local or remote) 5 = member of owner's community 6 = local or remote user having same ID as owner 7 = local super ID only Refer to the *GUARDIAN Operating System Programmer's Guide* or the *GUARDIAN Operating System Utilities Reference Manual* for an explanation of local and remote users, communities, and so forth. <param2> is not used with function 1.
2	**Disc:** Set file owner ID <param1>.<0:7> = group ID .<8:15> = user ID <param2> is not used with function 2.

SETMODE Functions Table (Continued)

<function>	
3	**Disc:** Set WRITE verification <param1>.<15> = 0 verified WRITEs off (default). = 1 verified WRITEs on. <param2> is used with DP2 disc files only. <param2> = 0 change the open option setting of the verify WRITEs option (default). = 1 change the file label default value of the verify WRITEs option.
4	**Disc:** Set lock mode <param1>.<15> = 0 default mode, suspends the process when lock or READ is attempted. = 1 alternate mode, rejects a lock or READ attempt with file system error 73 (file or record is locked). <param2> is not used with function 4.
5	**Line Printer:** Set system automatic perforation skip mode (assumes standard VFU function in channel 2) <param1>.<15> = 0 off, 66 lines per page = 1 on, 60 lines per page (default) For the 5530 line printer <param1> = 0 disable automatic perforation skip. = 1 enable automatic perforation skip (default). <param2> is not used with function 5.
6	**Line Printer or Terminal:** Set system spacing control <param1>.<15> = 0 no space = 1 single space (default setting) <param2> is not used with function 6.
7	**Terminal:** Set system auto line feed after receipt of carriage return line termination (default mode is configured): <param1>.<15> = 0 LFTERM line feed from terminal or network (default) = 1 LFSYS system provides line feed after line termination by carriage return. <param2> is not used with function 7.
8	**Terminal:** Set system transfer mode (default mode is configured) <param1>.<15> = 0 conversational mode = 1 page mode <param2> sets the number of retries of I/O operations. **NOTE** <param2> is used with 6530 terminals only.

SETMODE Functions Table (Continued)

<function>	
9	**Terminal:** Set interrupt characters <param1>.<0:7> = character 1 .<8:15> = character 2 <param2>.<0:7> = character 3 .<8:15> = character 4 (Default for conversational mode is backspace, line cancel, end of file, and line termination. Default for page mode is page termination.)
10	**Terminal:** Set parity checking by system (default is configured) <param1>.<15> = 0 no checking = 1 checking <param2> is not used with function 10.
11	**Terminal:** Set break ownership <param1> = 0 means BREAK disabled (default setting). = <cpu,pin> means enable BREAK. Terminal access mode after BREAK is typed <param2> = 0 normal mode (any type file access is permitted) = 1 BREAK mode (only BREAK-type file access is permitted)
12	**Terminal:** Set terminal access mode <param1>.<15> = 0 normal mode (any type file access is permitted) = 1 BREAK mode (only BREAK-type file access is permitted) File access type <param2>.<15> = 0 normal access to terminal = 1 BREAK access to terminal
13	**Terminal:** Set system read termination on ETX character (default is configured) <param1> = 0 no termination on ETX = 1 termination on first character after ETX = 3 termination on second character after ETX <param2> is not used with function 13.
14	**Terminal:** Set system read termination on interrupt characters (default is configured) <param1>.<15> = 0 no termination on interrupt characters (that is, transparency mode) = 1 termination on any interrupt character <param2> is not used with function 14.
20	**Terminal:** Set system echo mode (default is configured) <param1>.<15> = 0 system does not echo characters as read. = 1 system echoes characters as read. <param2> is not used with function 20.
21	**Card reader:** Set system read mode <param1> = 0 set ASCII read mode (default). 1 set column-binary read mode. 2 set packed-binary read mode. <param2> is not used with function 21.

SETMODE Functions Table (Continued)

<function>	
22	**Line printer** (subtype 3, 4, 6, and 32) or **terminal:** Set baud rate <param1> = 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 **NOTE** The 5520 line printer supports only the 110, 150, 300, 600, 1200, 2400, 4800, and 9600 baud rates. The 5530 line printer supports only the 75, 150, 300, 600, 1200, 2400, 4800, and 9600 baud rates. If no baud rate is specified in SYSGEN, then 9600 baud is used. The default is what was specified at SYSGEN. The asynchronous controller supports only the 150, 300, 600, 1200, and 1800 baud rates. <param2> is not used with function 22.
23	**Terminal:** Set character size <param1> = 0 character size = 5 bits 1 character size = 6 bits 2 character size = 7 bits 3 character size = 8 bits <param2> is not used with function 23.
24	**Terminal:** Set parity generation by system <param1> = 0 parity = odd 1 parity = even 2 parity = none <param2> is not used with function 24.
25	**Line printer** (subtype 3): Set form length <param1> = length of form in lines <param2> is not used with function 25.
26	**Line printer** (subtype 3): Set or clear vertical tabs <param1> > = 0 is (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). <param2> is not used with function 26.

SETMODE Functions Table (Continued)

<function>	
27	**Line printer** or **terminal:** Set system spacing mode <param1>.<15> = 0 postspace (default setting) = 1 prespace <param2> is not used with function 27.
28	**Line printer** or **terminal:** Reset to configured values <param1> = 0 <param2> is not used with function 28. For the 5530 line printer, SETMODE 28 resets all the SETMODE parameters back to their SYSGEN values and also reinitializes the printer. **NOTE** SETMODE 29 (set auto answer or control answer mode) is the only SETMODE not affected by a SETMODE 28.
29	**Line printer** (subtype 3, 4, 6, or 32): Set automatic answer mode or control answer mode. <param1>.<15> = 0 CTRLANSWER 1 AUTOANSWER (default) The default is what was specified at SYSGEN. If no mode is specified at SYSGEN, then AUTOANSWER is used. <param2> is not used with function 29. **NOTE** SETMODE option 29 remains in effect even after the file is closed. SETMODE 29 is the only SETMODE not affected by a SETMODE 28. SETMODE 29 is not reset at the beginning of each new job. Therefore, you should always issue a SETMODE 29 at the beginning of your job to assure that you are in the desired mode (whether than in the mode left by the previous job).
30	**Allow nowait I/O operations to complete in any order** <param1>.<15> = 0 no (default) = 1 yes **NOTE** If <param1>.<15> is set to 1, nowait operations do not necessarily complete in the order the I/O process returns them. (Operations that complete in time for AWAITIO to collect their output return in the order issued.) <param2> is not used with function 30.
31	**Set packet mode** <param1>.<0> = 0 ignore <param2> = 1 <param2> specifies leading packet size. <param2> = 0 use default packet size for transmission (default). >0 is size of first outgoing packet in each WRITE OR WRITEREAD request. It must be smaller than configured packet size.

SETMODE Functions Table (Continued)

<function>	
32	**Set X.25 call setup parameters** <param1>.<0> = 0 do not accept charge. = 1 accept charge. .<1> = 0 do not request charge. = 1 request charge. .<2> = 0 is normal outgoing call. = 1 is priority outgoing call. .<8:15> = port number (0-99) To determine the actual value for port number, refer to specifications on your own network.
33	**Seven-track tape drive:** Set conversion mode <param1> = 0 ASCIIBCD (even parity) (default) = 1 BINARY3TO4 (odd parity) = 2 BINARY2TO3 (odd parity) = 3 BINARY1TO1 (odd parity)
36	**Allow requests to be queued on $RECEIVE based on process priority** <param1>.<15> = 0 use first-in-first-out (FIFO) ordering (default). = 1 use process priority ordering. <param2> is not used with function 36.
37	**Line printer** (subtype 1, 4, 5, or 6): Get device status <param1> is not used with function 37. <param2> is not used with function 37. <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; 1 = paper moving ** RESERVED FOR LATER USE ** .<11> = BOF, Bottom of form — 0 = not at BOF; 1 = at bottom .<12> = TOF, Top of form — 0 = not at top; 1 = at top .<13> = DPE, Device parity error — 0 = parity OK; 1 = parity error .<14> = NOL, Not on line — 0 = on line; 1 = not on line .<15> = NRY, Not ready — 0 = ready; 1 = not ready All other bits are undefined.

SETMODE Functions Table (Continued)

<function>

NOTE

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, CID must be checked when PMO, BOF, and TOF are tested, since the old cable version does not return any of these states.

<last-params> for printer (subtype 4)

<last-params>[0] = primary status returned from printer:

.<9:11> = full status field	0 = partial status
	1 = full status
	2 = full status / VFU fault
	3 = reserved for future use
	4 = full status / data parity error
	5 = full status / buffer overflow
	6 = full status / bail open
	7 = full status / auxiliary status available
.<12> = buffer full	0 = not full
	1 = full
.<13> = paper out	0 = OK
	1 = paper out
.<14> = device power on	0 = OK
	1 = POWER ON error
.<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; 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.

<last-params> for printer (subtype 6)

<last-params>[0] contains the primary status.
[1] contains the auxiliary status.

SETMODE Functions Table (Continued)

<function>

Primary status bits are:

[0].<0:8> = 0 undefined
.<9> = 1 reserved
.<10:12> = 0 no faults
= 1 printer idle
= 2 paper out
= 3 end of ribbon
= 4 data parity error
= 5 buffer overflow
= 6 cover open
= 7 auxiliary status available
.<13> = 0 buffer not full
= 1 buffer full
.<14> = 0 OK
= 1 device power on error
.<15> = 0 OK
= 1 device not ready

If primary status <last-params>[0].<10:12> = 7; auxiliary status word is:

[1].<0:7> = undefined
.<8:11> = fault display status (most significant HEX digit)
.<12:15> = fault display status (least significant HEX digit)

Fault Display Status Summary

Operator Display	Aux Status .<8:11>	Aux Status .<12:15>	Problem Description
None	0	0	No faults
E01	0	1	Paper out
E03	0	3	Cover open
E06	0	6	End of ribbon
E07	0	7	Break
E11	1	1	Parity error
E12	1	2	Unprintable character
E22	2	2	Carrier loss
E23	2	3	Buffer overflow
E30	3	0	Printwheel motor fault
E31	3	1	Carriage fault
E32	3	2	Software fault
E34	3	4	Hardware fault

38 **Terminal:** Set special line termination mode and character

<param1> = 0 sets special line termination mode. <param2> is the new line termination character. The line termination character is not counted in the length of a READ. No carriage return or line feed is issued (the cursor is not moved) at the end of a READ.

= 1 sets special line termination mode. <param2> is the new line termination interrupt character. The line termination character is counted in the length of a READ. No carriage return or line feed is issued (the cursor is not moved) at the end of a READ.

= 2 resets special line termination mode. The line termination interrupt character is restored to its configured value. <param2> must be present but is not used.

SETMODE Functions Table (Continued)

<function>	
	<param2> = the new line termination interrupt character if <param1> = 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].
50	**Enable/disable 3270 COPY** <param1> = 0 suppress COPY = 1 allow COPY
51	**Get/set 3270 status** <param1> status flags mask to set <param2> is not used with function 51. Refer to the *Device Specific Access Methods—AM3270/TR3271 Manual* for the flags mask information.
52	**Tape drive:** Set short WRITE mode <param1> = 0 allow WRITEs shorter than 24 bytes (default) = 1 disallow WRITEs shorter than 24 bytes <param2> is not used with function 52. **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.
53	**Enable/disable receipt of status** <param1> = 0 disable status receive = 1 enable status receive <param2> = the response ID
54	**Return control unit and device assigned to subdevice** <param1> is not used with function 54. <param2> is not used with function 54. <last-params>[0].<0:7> = 0 .<8:15> = subdevice number known by AM3270 [1].<0:7> = standard 3270 control-unit address .<8:15> = standard 3270 device address
59	**Return count of bytes read** <param1> = count of actual bytes read <param2> = 0
57	**Disc:** Set serial WRITEs option (overrides SYSGEN setting for this file). <param1> = 0 system automatically selects serial or parallel WRITEs (default). = 1 allow only serial WRITEs to mirrored volumes. <param2> is used with DP2 disc files only. <param2> = 0 change the open option setting of the serial WRITEs option (default). = 1 change the file label default value of the serial WRITEs option.

SETMODE Functions Table (Continued)

<function>	
66	**Tape drive** (subtype 2): Set density <param1> = 0 800 bpi (NRZI) = 1 1600 bpi (PE) = 2 6250 bpi (GCR) = 3 as indicated by switches on tape drive <param2> is not used with function 66.
67	**AUTODCONNECT for full-duplex modems:** Monitor carrier detect or data set ready <param1> = 0 disable AUTODCONNECT (default setting). 1 enable AUTODCONNECT. <param2> is not used with function 67.
68	**Line printer** (subtype 4): Set horizontal pitch <param1> = 0 normal print (default) = 1 condensed print = 2 expanded print <param2> is not used with function 68.
70	**Permit nowait I/O operations to reference or modify user buffer** <param1>.<15> = 0 reference to user buffer permitted = 1 reference to user buffer not permitted between file system calls <param2> is not used with function 70.
90	**Disc:** Set buffered option defaults are same as CREATE <param1> = 0 buffered = 1 WRITE-thru <param2> is used with DP2 disc files only. <param2> = 0 change the open option setting of the buffered option (default). = 1 change the file label default value of the buffered option.
91	**Disc:** Set ACCESSTYPE open option (<function> 91 is not applicable for alternate-key files on DP2 disc files.) <param1> = 0 system managed (default) = 1 direct I/O, bypass disc cache = 2 random access, LRU-chain buffer = 3 sequential access, reuse buffer <param2> is not used with function 91.
92	**Disc:** Set maximum number of extents for a nonpartitioned file. (<function> 92 is not applicable for alternate-key files on DP2 disc files.) <param1> = new maximum number of extents value, defaults to 16 extents <param2> is not used with function 92.
93	**Disc:** Set buffer length for an unstructured file <param1> = new BUFFERSIZE value, must be valid DP2 blocksize. Valid DP2 block-sizes are 512, 1024, 2048, 4096 bytes (the default is 4096 bytes). <param2> is not used with function 93.

SETMODE Functions Table (Continued)

<function>	
94	**Disc:** Set audit-checkpoint compression option for an audited file. <param1> = 0 no audit-checkpoint compression (default) = 1 audit-checkpoint compression enabled <param2> = 0 change the open option setting of the audit-checkpoint compression option (default). = 1 change the file label default value of the audit-checkpoint compression option.
95	**Disc:** Flush dirty cache buffers (WRITE access required) <param1> is not used with function 95. <param2> is not used with function 95. <last-params> if specified, SETMODE returns: 0 = broken file flag off after all dirty cache blocks were written to disc 1 = broken file flag on, indicating some part of file is bad, possibly due to failed WRITE of a dirty cache block
110	**Set Shift In/Shift Out** (SISO) code extension technique for an individual subdevice **NOTE** SETMODE 110 is supported in AM6520 for CRT protocol when used for 6530 terminals, ITI protocol for 6530 terminals, and PTR protocol for 5520 printers. <param1> = 0 disable SISO (default setting) = 1 enable SISO <param2> is not used with function 110.
113	**Set screen size** <param1> = 1 the screen width (40, 66, 80, or 132) <param2> = the screen length (25, 28) <last-params> is an integer reference parameter containing two elements: <last-params>[0] = old <param1> [1] = old <param2>

SECTION D

SYSTEM PROCEDURE CALLS SYNTAX

This appendix contians a syntax summary of all the system procedure calls that are described in this manual.

```
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
                  ,<in-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
                                ,[ <spare1> ]                      ! i
                                ,[ <spare2> ]                      ! i
                                ,[ <spare3> ]                      ! i
                                ,[ <spare4> ]      ! error if used
                                ,[ <spare5> ]  );  ! error if used


{ <status> := }  SORTMERGESTATISTICS ( <ctlblock>          ! i
{ CALL         }                      ,<length>            ! i, o
                                      ,<statistics>        ! o
                                      ,[ <spare1> ]    ! 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
```

```
{ <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 E

ENFORM ERRORS

This appendix contains a list of the ENFORMRECEIVE and ENFORMSTART error codes and their meanings.

ENFORMRECEIVE

If an error occurs during the execution of the ENFORMRECEIVE procedure, the number of the error returns in <error-number>. Any of these error conditions terminates the ENFORM program. If the query processor is dedicated, it is deleted.

Some ENFORMRECEIVE errors have additional information written to the receiving buffer specified by the <buffer> parameter. The <buffer-length> parameter for ENFORMSTART specifies the number of bytes of error information written to <buffer>. The additional error message has a maximum length of 30 bytes. If <buffer-length> is less than 30 bytes, the error information is truncated to whatever length was specified in <buffer-length>.

ENFORMSTART

If an error occurs during the execution of the ENFORMSTART procedure, the number of the error returns to <error-number>. Any of these error conditions terminates the query processor. If the query processor is dedicated, it is deleted.

ENFORM Error Number	ENFORMRECEIVE	ENFORMSTART
1	A query processor received a message out of sequence.	Not applicable
3	An error occurred during a system READ. Contents of the <buffer> parameter are: • File System Number High-order byte of second word is in binary format. • File Name (internal format) Next 12 words contain the name, in internal format, of the physical file associated with the error.	An error occurred in trying to READ the <compiled-physical-filename>.
4	An error occurred during file system WRITE. Contents of the <buffer> parameter are: • File System Number High-order byte of second word is in binary format. • File Name (internal format) Next 12 words contain the name, in internal format, of the physical file associated with the error.	Not applicable
5	An error occurred during file system POSITION or KEYPOSITION. Contents of the <buffer> parameter are: • File System Number High-order byte of second word is in binary format.	

ENFORM Error Number	ENFORMRECEIVE	ENFORMSTART
	• File Name (internal format) Next 12 words contain the name, in internal format, of the physical file associated with the error.	Not applicable
6	An error occurred during file system CONTROL. Contents of the <buffer> parameter are: • File System Number High-order byte of second word is in binary format. • File Name (internal format) Next 12 words contain the name, in internal format, of the physical file associated with the error.	Not applicable
7	An error occurred during SORT. Not applicable Contents of the <buffer> parameter are: • File System Number High-order byte of second word is in binary format. • File Name (internal format) Next 12 words contain the name, in internal format, of the physical file associated with the error. • File Error Code Next word is in binary format.	Not applicable
8	An error occurred during file system OPEN. Contents of the <buffer> parameter are: • File System Number High-order byte of second word is in binary format. • File Name (internal format) Next 12 words contain the name, in internal format, of the physical file associated with the error.	An error occurred during file system OPEN. The error occurred while trying to open the <compiled-physical-filename>.
9	An error occurred during file CREATE.	Not applicable
10	Strategy exceeded the value of the @COST-TOLERANCE option variable. Actual ENFORM strategy cost for the query (number 1-8) is: High-order byte of second word is in binary format.	Not applicable

ENFORM Error Number	ENFORMRECEIVE	ENFORMSTART
11	An attempt was made to divide by 0.	Not applicable
12	A query contains an illegal combination of links. • Error Type High-order byte of second word is in binary format. 0 = ENFORM error 92. At least one record has no LINK or WHERE clause relating it to any other record. Query is not processed because a cross-product results. 1 = ENFORM error 178. The record on the right side of a link optional is linked back to the record on the left side. 2 = ENFORM error 179. A record appears on the right side of more than one link optional. Refer to the *ENFORM Reference Manual* for a complete explanation of error types 2 and 3. Refer to the *ENFORM User's Guide* for more information about ENFORM errors.	Not applicable
14	The read limit exceeds the value of the @READS option variable.	Not applicable
15	Not applicable	A required ENFORMSTART parameter is missing.
16	Not applicable	Some failure occurred in creation or in communicating with a query processor.
17	Not applicable	Either the <param-list> or <assign-list> is not in the correct format.
18	Not applicable	The value specified for <buffer-length> is less than 6.
19	Not applicable	A restart (restart flag not equal to zero) call made to ENFORMSTART was done under the following invalid conditions: • There was no previous successful call to ENFORMSTART. • ENFORMFINISH called before this ENFORMSTART. • Last ENFORMRECEIVE did not end in end-of-file or error status.

ENFORM Error Number	ENFORMRECEIVE	ENFORMSTART
20	Not applicable	The file named by <compiled-physical-filename> cannot be executed. Either the physical file has an invalid file code for a compiled query, or the compiled query contains a LIST statement rather than a FIND statement.
21	Not applicable	The file named by <compiled-physical-filename> has an outdated version number.
22	Not applicable	The amount of time specified by the timeout parameter elapsed with no response from the query processor.
23	<Ctlblock> was modified by the host application program since the last call to the ENFORM procedures.	<Ctlblock> was modified by the host application program since the last call to the ENFORM procedure.
24	Not applicable	The amount of stack space needed to build the message being sent to the query processor is not available.
25	There was a failure relating to the use of an ENFORM server (process file). 0 = Error returned from server. Server error name and server error number contain a file name and an error number supplied by the server. 1-5 = Error returned from query processor. Server error name contains the file name of the server that caused the error. Server error number is not set. 1 = illegal dictionary description (ENFORM error 112) 2 = illegal use of KEY item (ENFORM error 58) 3 = illegal LINK field (ENFORM error 68) 4 = insufficient memory for buffer (ENFORM error 50) 5 = incorrect reply length (ENFORM error 114). • Server error name Next 12 words contain a file name, in internal format, associated with the server. • Server error number Next word is in binary format.	Not applicable

APPENDIX F

INTERPROCESS SYSTEM MESSAGES

This appendix lists all system messages that are sent to processes. The GUARDIAN operating system sends the following messages to other processes through the $RECEIVE file. These messages do not appear on the terminal but are sent and received by processes alone. For each system message, its name, internal form, and meaning are described.

For further information about system messages that are sent to processes, refer to the GUARDIAN Operating System Programmer's Guide.

The completion of a read associated with a system message returns a condition code of greater than (CCG) and error 6 from FILEINFO.

NOTE

Like all interprocess messages, system messages read (by calling the READUPDATE procedure) must be replied to in a corresponding call to REPLY. If the application process is performing message queueing, LASTRECEIVE or RECEIVEINFO must also be called immediately following completion of the READUPDATE, and the message tag must be passed back to the REPLY procedure.

The first word of a system message is a number between -2 and -35. Following are the system messages and their formats in word elements.

-2 CPU DOWN

cause: GUARDIAN has failed to receive an "I'm alive" message from the specified processor.

system action: There are two forms of the CPU DOWN message. GUARDIAN sends the following form to a process if a processor module it is monitoring with the MONITORCPUS procedure fails:

```
<sysmsg>            = -2
<sysmsg>[1]         = CPU
```

GUARDIAN sends the second form to an ancestor process when it deletes the indicated process name from the process-pair directory because of a processor module failure:

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

This means that the named process (pair) no longer exists.

recovery: Corrective action, if any, is application dependent.

-3 CPU UP

cause: A processor module being monitored with the MONITORCPUS procedure was reloaded.

system action: GUARDIAN sends a message of the following form:

```
<sysmsg>            = -3
<sysmsg>[1]         = CPU
```

recovery: Corrective action, if any, is application dependent.

-5 PROCESS NORMAL DELETION (STOP)

cause: Process deletion was due to a call to the process control STOP procedure.

system action: There are two forms of the STOP message. GUARDIAN sends the following form to the deleted process's creator if the deleted process was not named or to one member of a process pair when it deletes the other:

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

GUARDIAN sends the second form to a process pair's ancestor when it deletes the process name from the process-pair directory. This indicates that neither member of the process pair exists:

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

recovery: Corrective action, if any, is application dependent.

-6 PROCESS ABNORMAL DELETION (ABEND)

cause: Process deletion was due to a call to the process control ABEND procedure, or because the deleted process encountered a trap condition and was aborted by GUARDIAN.

system action: There are two forms of the ABEND message. GUARDIAN sends the first form to a deleted process's creator if the deleted process was not named or to one member of a process pair when the other member is deleted:

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

GUARDIAN sends the second form to a process pair's ancestor when it deletes the process name from the process-pair directory. This indicates that neither member of the process pair exists:

```
<sysmsg>              = -6
<sysmsg>[1] FOR 3     = $<process name> of deleted
                        process (pair)
<sysmsg>[4]           = -1
```

recovery: Corrective action, if any, is application dependent.

-8 CHANGE IN STATUS OF NETWORK NODES

cause: The process was running on a system that is part of a network and has enabled receipt of remote status change messages by passing "1" as a parameter to the MONITORNET procedure.

system action: GUARDIAN sends the process the following message:

```
<sysmsg>              = -8
<sysmsg>[1].<0:7>     = system number
<sysmsg>[1].<8:15>    = number of CPUs
<sysmsg>[2]           = current processor-status
                        bit mask
<sysmsg>[3]           = previous processor-status
                        bit mask
```

recovery: Corrective action, if any, is application dependent.

-10 SETTIME

cause: The system manager or operator reset the internal clock of the indicated CPU.

system action: GUARDIAN sends the process the following message, provided the process has enabled receipt of new messages by a call to MONITORNEW:

```
<sysmsg>            = -10
<sysmsg>[1]         = CPU
```

recovery: Corrective action, if any, is application dependent.

-11 POWER ON

cause: The indicated processor had a POWER OFF, then a POWER ON condition.

system action: GUARDIAN sends the process the following message, provided the process has enabled receipt of new messages by a call to MONITORNEW:

```
<sysmsg>            = -11
<sysmsg>[1]         = CPU
```

recovery: Corrective action, if any, is application dependent.

-12 NEWPROCESSNOWAIT COMPLETION

cause: A call to the NEWPROCESSNOWAIT procedure completed.

system action: GUARDIAN sends the process the following message:

```
<sysmsg>            = -12
<sysmsg>[1]         = error
<sysmsg>[2] FOR 2   = tag
<sysmsg>[4] FOR 4   = process ID
```

recovery: Corrective action, if any, is application dependent.

-20 BREAK RECEIVED FROM TERMINAL

cause: The BREAK key was pressed on a terminal being monitored.

system action: GUARDIAN sends the process the following message, provided the process has specified break monitoring through a call to SETMODE or SETMODENOWAIT:

<sysmsg>	= - 20
<sysmsg>[1]	= logical device number, in binary, of device where BREAK was pressed
<sysmsg>[2]	= system number, in binary, of logical device number

recovery: Corrective action, if any, is application dependent.

-21 3270 DEVICE STATUS RECEIVED

cause: A call to SETMODE 53 was made by the application to monitor subdevice status, pass the information to TR3271 by way of a call to SETMODE 51, and issue this status message.

system action: GUARDIAN sends the process the following message:

<sysmsg>	= -21
<sysmsg>[1]	= the response ID
<sysmsg>[2]	= the actual 3271 status bytes, in which the sense byte = <0:7> and the status byte = <8:15>
<sysmsg>[3]	= a translation of the device status to status bits. The application may pass this word directly to TR3271 by way of SETMODE 51 to post the status on a TR3271 subdevice.

recovery: Corrective action, if any, is application dependent.

-22 TIME SIGNAL

cause: A timer set by a call to SIGNALTIMEOUT timed out.

system action: GUARDIAN sends the process the following message:

<sysmsg>	= -22
<sysmsg>[1]	= <parameter1> supplied to SIGNALTIMEOUT (0 if none)
<sysmsg>[2] FOR 2	= <parameter2> supplied to SIGNALTIMEOUT (0D if none)

recovery: Corrective action, if any, is application dependent.

-23 MEMORY LOCK COMPLETION

cause: A call to LOCKMEMORY waited for memory but completed successfully before the specified time limit was reached.

system action: GUARDIAN sends the process the following message:

<sysmsg>	= -23
<sysmsg>[1]	= <parameter1> supplied to LOCKMEMORY (if none supplied, 0)
<sysmsg>[2] FOR 2	= <parameter2> supplied to LOCKMEMORY (if none supplied, 0D)

recovery: Corrective action, if any, is application dependent.

-24 MEMORY LOCK FAILURE

cause: A call to LOCKMEMORY waited for memory and timed out without completing the lock.

system action: GUARDIAN sends the process the following message:

<sysmsg>	= -24
<sysmsg>[1]	= <parameter1> supplied to LOCKMEMORY (if none supplied, 0)
<sysmsg>[2] FOR 2	= <parameter2> supplied to LOCKMEMORY (if none supplied, 0D)

recovery: Corrective action, if any, is application dependent.

-26 TIME SIGNAL

cause: A timer set by a call to SIGNALPROCESSTIMEOUT timed out.

system action: GUARDIAN sends the process the following message:

<sysmsg>	= -26
<sysmsg>[1]	= <parameter1> supplied to SIGNALPROCESSTIMEOUT (0 if none)
<sysmsg>[2] FOR 2	= <parameter2> supplied to SIGNALPROCESSTIMEOUT (0D if none)

recovery: Corrective action, if any, is application dependent.

-30 PROCESS OPEN

cause: A process was opened by another process. This message is also received if the OPEN was by the backup process of a process pair. A process can therefore expect two of these messages when opened by a process pair.

system action: GUARDIAN sends the process the following message, provided the process has opened its $RECEIVE file with <flags>.<1> = 1:

<sysmsg>	= -30
<sysmsg>[1]	= <flags> parameter to caller's OPEN
<sysmsg>[2]	= <sync-or-receive-depth> parameter to caller's OPEN
<sysmsg>[3] FOR 4	= 0 if normal OPEN, process ID of primary process if an open by a backup process
<sysmsg>[7]	= 0 if normal OPEN, negative of the file number of file if an open by a backup process
<sysmsg>[8]	= process accessor ID of opener
<sysmsg>[9] FOR 4	= optional first qualified name of named process or blanks
<sysmsg>[13] FOR 4	= optional second qualified name of named process or blanks

recovery: Obtain the process ID of the opener by a subsequent call to LASTRECEIVE or RECEIVEINFO. Corrective action, if any, is application dependent.

-31 PROCESS CLOSE

cause: Another process closed the receiver process. The closing process can also be the backup process of a process pair. Therefore, a process can expect two of these messages when being closed by a process pair.

system action: GUARDIAN sends the process the following message, provided the process has opened its $RECEIVE file with <flags>.<1> = 1:

<sysmsg>	= -31

recovery: Obtain the process ID of the closer by a subsequent call to LASTRECEIVE or RECEIVEINFO. Corrective action, if any, is application dependent.

-32 PROCESS CONTROL

cause: Another process called the CONTROL procedure, referencing the receiver process file.

system action: GUARDIAN sends the process the following message, provided the process has opened its $RECEIVE file with <flags>.<1> = 1:

<sysmsg>	= -32
<sysmsg>[1]	= <operation> parameter to caller's CONTROL
<sysmsg>[2]	= <parameter> parameter to caller's CONTROL

recovery: Obtain the process ID of the caller to CONTROL by a subsequent call to LASTRECEIVE or RECEIVEINFO. Corrective action, if any, is application dependent.

-33 PROCESS SETMODE

cause: Another process called the SETMODE or SETMODENOWAIT procedure, referencing the receiver process file.

system action: GUARDIAN sends the process the following message, provided the process has opened its $RECEIVE file with <flags>.<1> = 1:

<sysmsg>	= -33
<sysmsg>[1]	= <function> parameter to caller's SETMODE or SETMODENOWAIT
<sysmsg>[2]	= <parameter 1> parameter to caller's SETMODE or SETMODENOWAIT
<sysmsg>[3]	= <parameter 2> parameter to caller's SETMODE or SETMODENOWAIT

recovery: Obtain the process ID of the caller to SETMODE or SETMODENOWAIT by a subsequent call to LASTRECEIVE or RECEIVEINFO. Corrective action, if any, is application dependent.

-34 PROCESS RESETSYNC

cause: Another process called the RESETSYNC procedure, referencing the receiver process file. A call to the CHECKPOINT procedure might contain an implicit call to RESETSYNC.

system action: GUARDIAN resets the sync ID value for that file to 0 and sends the process the following message, provided the process has opened its $RECEIVE file with <flags>.<1> = 1:

<sysmsg>	= -34

recovery: Obtain the process ID of the caller to RESETSYNC by a subsequent call to LASTRECEIVE or RECEIVEINFO. A server process using the sync ID mechanism should clear its local copy of the sync ID value. Corrective action, if any, is application dependent.

-35 PROCESS CONTROLBUF

cause: Another process called the CONTROLBUF procedure, referencing the receiver process file.

system action: GUARDIAN sends the process the following message, provided the process has opened its $RECEIVE file with <flags>.<1> = 1:

<sysmsg>	= -35
<sysmsg>[1]	= <operation> parameter to caller's CONTROLBUF
<sysmsg>[2]	= <count> parameter to caller's CONTROLBUF
<sysmsg>[3] FOR n	= <buffer> data from caller's CONTROLBUF, where n is number of words in <buffer>

recovery: Obtain the process ID of the caller to CONTROLBUF by a subsequent call to LASTRECEIVE or RECEIVEINFO. Corrective action, if any, is application dependent.

APPENDIX G

SORT/MERGE ERRORS

This appendix lists the SORT/MERGE error codes.

Error Code <errnum>	SORT/MERGE Error Messages
1	The 'CTLBLOCK' parameter to SORTMERGESTART is required.
2	The 'KEYS' parameter to SORTMERGESTART is required.
3	The number of key fields must be 1 to 63 inclusive.
4	An error has prevented creation of the SORT process.
5	Communications with the SORT process have failed.
6	The SORT process has stopped unexpectedly.
7	SORTMERGESEND was called unexpectedly.
8	SORTMERGERECEIVE was called unexpectedly.
9	Record length to SORTMERGESEND is too small or large.
10	SORTMERGEFINISH was called unexpectedly.
11	The free list file cannot be opened.
12	Invalid flag or combination of flags.
20	Communications with SORTPROG have broken down.

Error Code <errnum>	SORT/MERGE Error Messages
21	Communications with SORTPROG were garbled.
22	The memory space for sorting is insufficient.
23	The from file could not be opened.
24	A temporary 'TO FILE' is too small.
25	One of the key fields is of an undefined type.
26	A key-field location exceeds the record size.
27	The 'TO FILE' already exists and cannot be purged.
28	A scratch file cannot be opened.
29	A WRITE to the 'TO FILE' has failed.
30	A WRITE to a scratch file has failed.
31	A READ from the 'FROM FILE' has failed
32	A READ from a scratch file has failed.
33	A CONTROL operation has failed.
34	An EDITREAD has failed from the 'FROM FILE.'
35	Creation of a scratch file has failed.
36	A POSITION has failed in a scratch file.
37	Creation of the 'TO FILE' has failed.
38	The 'TO FILE' could not be opened.
39	An input record exceeded the record size.
40	Configuration problem: no I/O buffer space.
41	Configuration problem: no SHORTPOOL.
42	The MEM size must be in the range 1 to 64.

Error Code <errnum>	SORT/MERGE Error Messages
43	The PRIORITY must be in the range 1 to 200.
44	Invalid control block; procedure call rejected.
45	Invalid scratch file block size.
46	Real number keys must be word-aligned.
47	SORTMERGESTATISTICS was called unexpectedly.
48	A signed ASCII numeric key is larger than 32 bytes.
49	Invalid exclusion mode specified.
50	EDIT files cannot be 'TO FILES.'
51	Invalid file type specified for 'TO FILE.'
52	Only one file can be sorted by SORTMERGESEND.
53	A 'TO FILE' cannot be a file to be merged.
54	Invalid scratch file name.
55	Too many from files specified.
56	Invalid number of files to be sorted or merged.
57	Collating sequence table must be present.
58	SORTMERGESTART was called unexpectedly.
59	An input record is too small.
61	Spare parameters cannot be present.
62	Key length must be greater than zero.
63	Reserved flags cannot be set.

APPENDIX H

RESERVED PROCESS NAMES

This appendix contains the names that should be avoided when choosing process names. The names listed here are reserved for Tandem use:

```
$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
```

INDEX

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

$

'

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

System Procedure Calls Reference Manual

NonStop™ Systems

82359 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