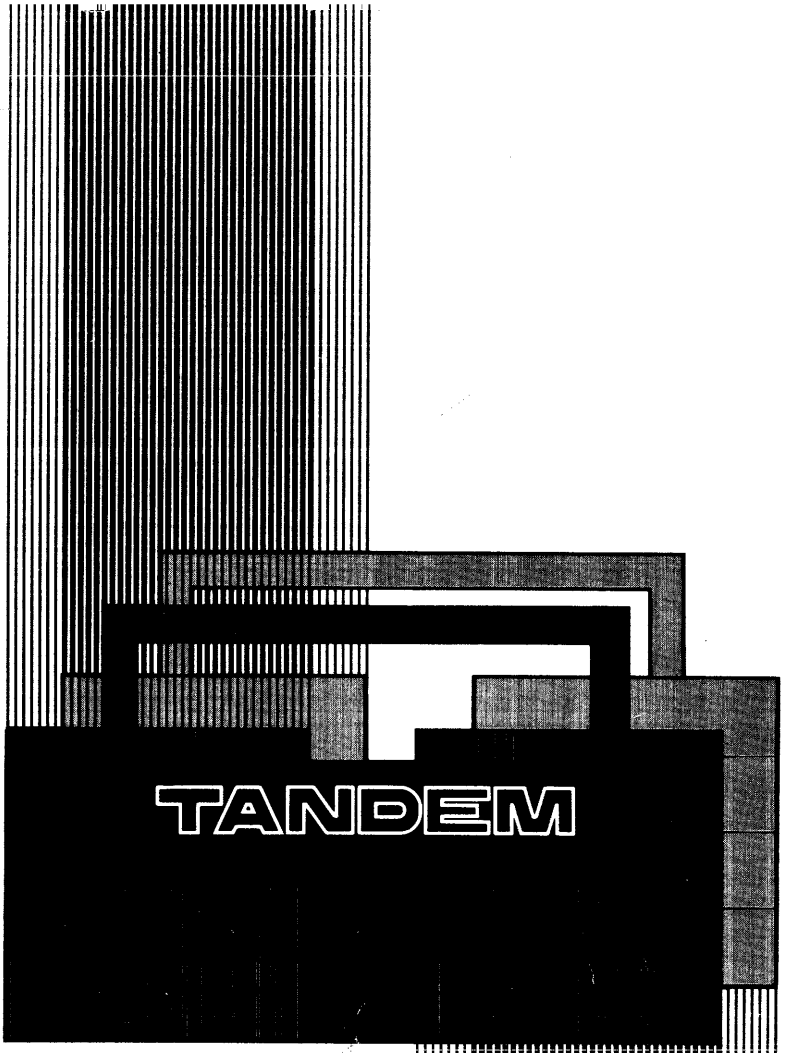


GUARDIAN Operating System Programming Manual

Volume 2

GUARDIAN (TM) OPERATING SYSTEM

PROGRAMMING MANUAL

Volume 2

Tandem Computers Incorporated
19333 Vallco Parkway
Cupertino, California 95014

P/N 82337 A00

April 1982
Printed in U.S.A.

The GUARDIAN Operating System Programming Manual was published in October 1981 as a two-volume manual with part number 82096 A00 for each volume. The 82096 A00 version was the first edition to include both NonStop and NonStop II information in one publication.

The April 1982 version revises the 82096 A00 version. With this revision, the two volumes have been given separate part numbers; volume I is now 82336 A00, and volume II is now 82337 A00. Users of this publication should obtain both volumes.

Information has been added on the following new devices:

5106 Tri-Density Tape Subsystem
4116 540MB Winchester Disc Drive
4110/4111 128MB Winchester Disc Drives
5513/5514 Band Line Printers
5520 Serial Matrix Printer
6530 Multi-page Terminal

PREFACE

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

Specifically, the manual discusses:

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

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

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

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

Because of its size, this manual is bound as two volumes. Volume 1 covers:

- Introduction
- File System
- Process Control

Volume 2 covers:

- Utility Procedures
- Checkpointing Facility
- Traps
- Security System
- Memory Management Procedures
- Sequential I/O Procedures
- Formatter
- COMINT/Application Interface
- NonStop Programming Example
- Appendices

To simplify cross-referencing and indexing, the two volumes are section-numbered as if they were one book; i.e., volume 2 begins with section 4. Each volume contains a complete table of contents and index for both volumes, so that the reader can easily find the information he needs.

Information that applies only to NonStop systems (not to NonStop II systems) and information that applies only to NonStop II systems is clearly marked as such -- for instance, by a page header or figure title stating "(NonStop systems only)" or by a sentence at the start of a paragraph beginning "On NonStop II systems, ...". Material not so marked applies to both types of system.

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

For NonStop systems only:

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

For NonStop II systems only:

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

For both systems:

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

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

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

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

- Technical Communications Library

CONTENTS

Volume 1

FIGURES

Volume 1

Volume 2

TABLES

Volume 1

Volume 2

SYNTAX CONVENTIONS IN THIS MANUAL

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

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

form, the separator is to be entered before each repetition, not before the first occurrence of the item or series.)

Colons

A colon (:) between two syntactical entities signifies a "from...through..." relationship. For example, the command CHANGE 1:6 " " specifies that positions 1 through 6 are to be filled with spaces.

Punctuation

All punctuation and symbols other than those described above must be entered precisely as shown. If any of the above punctuation appears enclosed in quotation marks, that character is not a syntax descriptor but a required character, and must actually be entered.

System Procedure Calls

Calls to operating system procedures are shown in the following form:

```
{ CALL          } <procedure name> ( <parameters> )
{ <retval> := }
```

CALL is a TAL CALL statement.

"<retval> :=" indicates that the procedure is a function procedure (i.e., it returns a value of the indicated <type> when referenced in an expression).

<procedure name> is the name of the operating system procedure.

Required parts of the calling sequence are underlined. Optional parameters may be omitted, but placeholder commas "," must be present except for right-side omitted parameters.

A function procedure's return value is described as follows:

<retval>, <type>

<type> is INT or INT:32

Note that a function procedure can be called with a CALL statement. However, the return value will be lost.

<parameters> are described as follows:

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

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

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

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

"value" indicates a value parameter.

SECTION 4

UTILITY PROCEDURES

The GUARDIAN operating system provides a number of utility procedures for use by application programs. These procedures are as follows:

CONTIME	takes 48 bits of a time stamp and provides a date and time in internal machine representation
DEBUG	calls the debug facility
FIXSTRING	edits a string of characters based on information supplied in an editing template
HEAPSORT	sorts an array of equal-size elements in place
INITIALIZER	reads the startup message and, optionally, the assign and param messages to prepare global tables and initialize File Control Blocks (FCB's)
LASTADDR	provides the global ('G'[0] relative) address of last word in the caller's data area
NUMIN	converts the ASCII representation of a number into its binary equivalent
NUMOUT	converts the internal machine representation of a number to its ASCII equivalent
SHIFTSTRING	upshifts or downshifts alphabetic characters in a string
TIME	provides the current date and time in internal machine representation
TIMESTAMP	provides the current value of the processor clock where this application is running
TOSVERSION	provides an identifying letter and number indicating which version of the GUARDIAN operating system is running

CONTIME Procedure

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

The call to the CONTIME procedure is:

```
CALL CONTIME ( <date and time> , <t0> , <t1> , <t2> )
---- ------- - -------------- - ---- - ---- - ---- -

where

   <date and time>, INT:ref:7,

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

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

   <t0>, <t1>, <t2>, INT:value,

     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).

example:

   CALL CONTIME (time, t[0], t[1], t[2]);
```

For example, CONTIME can be used to convert the <last mod time> timestamp into a readable form:

```
INT last^t[0:2], date^time[0:6];
```

Then the last modification time is obtained through a call to the FILEINFO procedure:

```
CALL FILEINFO( fnum,,,,,,,,last^t);
```

Then CONTIME is used to convert the three words in "last^t" to a date and time:

```
CALL CONTIME(date^time,last^t,last^t[1],last^t[2]);
```

Seven words of date and time are returned in "date^time".

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

The call to the DEBUG procedure is:

```
CALL DEBUG
---- -----
```

For a description of the debug facility and instructions for using it, see the DEBUG Reference Manual for your type of system (NonStop system or NonStop II system).

The FIXSTRING procedure is used to edit a string based on subcommands provided in a template.

The call to the FIXSTRING procedure is:

```
CALL FIXSTRING ( <template> , <template length>
                 , <data> , <data length>
                 , <maximum data length>
                 , <modification status> )
```

where

<template>, STRING:ref,

is the character string to be used as a modification template.

<template length>, INT:value,

is the length, in bytes, of the template string.

<data>, STRING:ref,

is the string to be modified; the resulting string is returned in this parameter.

<data length>, INT:ref,

is the length, in bytes, of the data string. The length of the modified data string is returned in this parameter.

<maximum data length>, INT:value,

if present, contains the maximum length, in bytes, to which <data> may be expanded during the call to FIXSTRING. If omitted, 132 is used for this value.

<modification status>, INT:ref:1,

if present, is returned an integer value as follows:

0	no change was made to <data>.
1	a replacement, insertion, or deletion was performed on <data> (see "Considerations" below).

→

```
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
            length>.

example:

   CALL FIXSTRING ( template, temp^len, data, data^len,
                    mod^status );
   IF > THEN ...  ! too long
```

SUBCOMMANDS

There are three basic subcommands that may be used 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
```

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 to be 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"

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 (i.e., 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"

deletes the corresponding character in <data>.

I (or i), for "insert"

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 (i.e., are not ignored).

Examples:

replacement

```
<data> on entry to FIXSTRING:     THIS IS A STRING
<template>:                                rNEW STRING
<data> on return from FIXSTRING:  THIS IS A NEW STRING
```

implied replacement

```
<data> on entry to FIXSTRING:     THIS IS A STRIMG
<template>:                                    N
<data> on return from FIXSTRING:  THIS IS A STRING
```

replacement terminated by "//"

```
<data> on entry to FIXSTRING:     THID IS A BAD OLD STRING
<template>:                          S//    rNEW STRING    //
<data> on return from FIXSTRING:  THIS IS A NEW STRING
```

(If the first "//" had been omitted, the "r" would be considered part of the <replacement string> associated with the implied replacement that begins with "S", and the resulting string would be "THIS rNEW STRING". The second "//" could be omitted, but the four trailing blanks preceding it are necessary; without them, the resulting string would be "THIS IS A NEW STRINGRING".)

deletion

```
<data> on entry to FIXSTRING:    THIS IS A LONG STRING
<template>:                                 ddddd
<data> on return from FIXSTRING: THIS IS A STRING
```

(Note that one of the spaces surrounding "LONG" was deleted.)

"D" occurring as part of another subcommand

```
<data> on entry to FIXSTRING:     THIS IS XATA
<template>:                                rD
<data> on return from FIXSTRING:  THIS IS DATA
```

("D" is the <replacement string> associated with "r".)

insertion

```
<data> on entry to FIXSTRING:     THIS IS A STRING
<template>:                              i NEW
<data> on return from FIXSTRING:  THIS IS A NEW STRING
```

several operations combined in one <template>

```
<data> on entry to FIXSTRING:     THID IS A BAD OLD STRIG
<template>:                          S//    ddddrNEW//  iN
<data> on return from FIXSTRING:  THIS IS A NEW STRING
```

CONSIDERATIONS

- The <maximum data length> serves to protect data residing past the end of the <data> string. Therefore, <data> is truncated whenever <data length> exceeds <maximum data length> during processing by FIXSTRING.

  In particular, <data> is truncated by FIXSTRING if <data length> temporarily exceeds <maximum data length>, even if <template> contains delete subcommands that would have resulted in a <data> string of the correct length.

- If an insertion would cause the length of <data> to exceed <maximum data length>, the <insertion string> is truncated. Example:

  assume <maximum data length> is 6

  ```
  <data> on entry to FIXSTRING:     AB
  <template>:                        i1234567890
  <data> on return from FIXSTRING:  A1234B
  ```

- <modification status> is set to 1 if a replacement is performed that leaves <data> unchanged. For example:

  ```
  <data> on entry to FIXSTRING:    THIS IS A STRING
  <template>:                      THIS IS A STRING
  <data on return from FIXSTRING:  THIS IS A STRING
  ```

  <modification status> is set to 1, since a replacement has been performed by FIXSTRING.

USING FIXSTRING TO IMPLEMENT AN FC COMMAND

FIXSTRING is generally used to implement an FC command in an interactive process (for example, the debug facility or the GUARDIAN Command Interpreter).

An FC command could be implemented in an interactive command interpreter as follows:

```
  .
  .
INT .command[-1:3] := "< ",  ! command length <= 8 characters
    .last^command[0:3],     ! save previous command
     num,                   ! length of current command string
     save^num;              ! length of last command string

STRING .scommand := @command '<<' 1;  ! command addressed as string

  .
  .
INT PROC fc; FORWARD;
```

```
  .
  .
PROC command^interpreter;
  BEGIN
      .
      .
    INT repeat := 0;     ! a flag used to determine whether
                         ! command^interpreter should attempt
                         ! to execute a command upon return
                         ! from "fc"
      .
      .
    WHILE 1 DO    ! the main loop of command^interpreter executes
      BEGIN       ! until an "exit" command is encountered.

        IF NOT repeat THEN
          BEGIN
            command ':=' "<";  ! assume "<" is the prompt character
            CALL WRITEREAD(term, command, 1, 8, num);

               Displays the prompt character and reads a command,
               assuming "term" is the device number of the
               terminal.

          END;

        IF command = "FC" THEN repeat := fc
        ELSE
          BEGIN      ! identify and execute command,
                     ! or print an "illegal command" message.
            .
            repeat := 0;
          END;

        IF num THEN
          BEGIN
            save^num := num;
            last^command ':=' command FOR (save^num+1)/2;

               Saves last command and its length in case next
               command is "FC".

           END;
      END;          ! main loop
  END;              ! command^interpreter

  .
  .
INT PROC fc;
  BEGIN
    INT .temp^array[0:35],   ! array to hold modification template
        temp^len;            ! length of template
    STRING .s^temp^array := @temp^array '<<' 1;
                             ! temp^array addressed as string
```

```
        command[-1] ':=' "< ";
        num := save^num;
        command ':=' last^command FOR (num+1)/2;

        DO
          BEGIN

            CALL WRITE(term, command[-1], num + 2 );
                        ! display "<" followed by the last command.
            temp^array ':=' " .";  ! template prompt
            CALL WRITEREAD ( term, temp^array, 2, 72, temp^len );
                                   ! display prompt and read template.
            IF > OR temp^len = 2 AND temp^array = "//" THEN
              BEGIN      ! restore command
                num := save^num;
                command ':=' last^command FOR (num+1)/2;
                RETURN 0;
              END;

               An EOF or a template consisting of "//" causes "fc" to
               return 0, indicating that "command^interpreter" should
               not execute the command, but should prompt for a new
               command instead.  If the new command is "FC", then the
               string to be fixed is the command that was originally
               being modified on the previous call to "fc".

            CALL FIXSTRING ( s^temp^array, temp^len, scommand, num );

               "scommand" now contains the modified command, and "num"
               is its length.  If "temp^len" > 0, the loop executes
               again, displaying the modified command and expecting a
               new template.  If "temp^len" = 0, then a <cr> was input
               instead of a template.  In this case, FIXSTRING leaves
               the command unchanged and returns a value of 1,
               indicating that command^interpreter should attempt to
               identify and execute the command.

          END
        UNTIL NOT temp^len;  ! loop executes until "temp^len" = 0,
                             ! indicating a <cr>

        RETURN 1;    ! indicates to the calling procedure that
                     ! the command came from "fc" and should be
                     ! identified and executed.
      END;  !fc
```

The HEAPSORT procedure is used to sort an array of equal-size elements in place.

The call to the HEAPSORT procedure is:

```
CALL HEAPSORT ( <array> , <num elements> , <size of element>
---- -------- - ------- - -------------- - -----------------
                , <compare proc> )
                - -------------- -
```

where

<array>, INT:ref,

is an array containing equal-size elements to be sorted.

<num elements>, INT:value,

is the number of elements in <array> to be sorted.

<size of element>, INT:value,

is the size, in words, of each element in <array>.

<compare proc>, INT PROC,

is an application-supplied function procedure that is called by HEAPSORT to determine the sorted ordering (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> , .<element b> ;
   --- ------------ - ------------ -
```

where

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.

→

```
example:

   CALL HEAPSORT(array,num^elements,size,comp^proc);
```

The following example illustrates the use of HEAPSORT.

```
LITERAL element^size = 12;

INT .array[0:119], ! array to be sorted.
    num^elements;
```

Elements of twelve words each are to be sorted in ascending order. Therefore the following "compare proc" is written:

```
INT PROC ascending (a,b);
   INT .a, .b;
  BEGIN
    RETURN IF a < b FOR 12 THEN 1 ELSE 0;
  END;
```

Then HEAPSORT is called to sort "array":

```
    .
num^elements := 10;
CALL HEAPSORT(array,num^elements,element^size,ascending);
    .
```

sorts the ten elements in "array" in ascending order.

The INITIALIZER is a procedure used to read the startup and (optionally) assign and param messages sent by the Command Interpreter. The INITIALIZER procedure optionally prepares tables of a predefined structure and properly initialized file FCB's with the information read from the startup and assign messages.

The call to the INITIALIZER procedure is:

```
{ <status> := }  INITIALIZER  ( <rucb>
{ CALL        }  -----------  -
                                , <passthru>
                                , <startupproc>
                                , <paramsproc>
                                , <assignproc>
                                , <flags>      )
                                                -
```

where

<status>, INT,

is either:

0 = This is the primary process (of a potential process pair).

-1 = This is the backup process, CHECKMONITOR returned (it received no stack checkpoints from the primary), and bit 12 of <flags> was 1.

<rucb>, INT:ref,

is a table which contains pointers to the FCB's (see section 9, "Sequential I/O Procedures").

<passthru>, INT:ref,

is an array where the <startupproc>, <assignproc>, and <paramsproc> procedures may pass information back to the caller of the INITIALIZER.

<startupproc>, <paramsproc>, and <assignproc>

are application-supplied message processing procedures that are called by the INITIALIZER when a message of the appropriate type is received.

→

These procedures must be of the form:

```
PROC <name> ( <rucb>, <passthru>, <message>
---- ----  -       -           -

                ,<meslen> ,<match> )  VARIABLE
               -         -        -  --------
```

where

<rucb>, INT:ref,

is described in section 9, "Sequential I/O Procedures".

<passthru>, INT:ref,

is an array where the procedure may save information for the caller of the INITIALIZER.

<message>, INT:ref,

is the startup, the param, or one of the assign message(s) received. The maximum length of a message is 1028 bytes (including the trailing null characters).

<meslen>, INT:value,

is the length, in bytes, of the message.

<match>, INT:value,

is the number of FCB's whose entire logical file names match the logical file name in this ASSIGN message.

If this is not an assign message or if the <rucb> parameter is not passed, the match count is always zero.

<flags>, INT,

contains several fields that determine actions to be taken by the INITIALIZER, as follows:

```
<flags>.<0:10> =  must be zero
<flags>.<11>   =  request assign and param messages?
                    0 = yes      1 = no
```

→

```
<flags>.<12>    =  abend if backup takeover occurs before
                   first primary stack checkpoint?
                     0 = yes      1 = no

<flags>.<13>    =  if 1, CALL MONITORNET (-1)
<flags>.<14>    =  if 1, CALL MONITORCPUS (-1)
<flags>.<15>    =  if 1, CALL ARMTRAP (-1,-1)
```

The INITIALIZER procedure provides a way of receiving startup, assign, and param messages without concern for details of the $RECEIVE protocol. (See section 11, "COMINT/Application Interface".) The INITIALIZER obtains messages from $RECEIVE and calls the user-supplied procedure, passing the messages as a parameter to the procedure.

In addition, if the <rucb> parameter is supplied, the INITIALIZER will store FCB's based on the information supplied by the startup and assign messages. These FCB's are in the form expected by the sequential i/o procedures, and may be used with the sequential i/o procedures without change. If the application does not use the sequential i/o procedures to access the files, the information recovered from the assign messages may be obtained from the FCB's by using the SET^FILE procedure. See section 9, "Sequential I/O Procedures".

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 FCB's (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 is useful in program startup. It does the following:

In the primary process:

1. Inspects <flags>.<13:15>, and calls the appropriate procedures, if any.
2. Determines if this is a primary of the process pair.
3. Opens $RECEIVE.
4. Reads the startup sequence from MOM:
    a. Stores startup and assign information in <rucb> if an array was passed.

b. Calls procedures if any were passed (optionally, "assign" and "params" procedures).

c. Calls ABEND, if the messages that are read from $RECEIVE are not in the correct order. (The correct order is the startup message, then the assign messages, then the param message.)

d. Rejects messages from anyone other than MOM with reply code 100 (OPEN messages) or 60 (all others).

Note: If bit 11 of <flags> is 0, the INITIALIZER replies to MOM's startup message with an error return value of 70. This requests assign and param messages from MOM. If bit 11 of <flags> is 1, the INITALIZER replies to the startup message with an error return value of 0 and no reply text. This indicates that the process does not wish to receive assign or param messages. (See section 11, "Command Interpreter/ Application Interface".)

5. Closes $RECEIVE.

6. Performs the following:

   a. Substitutes the FCB's actual file names for default physical file names.

   b. Expands partial file names in the FCBs.

   c. Places information into the <rucb> if used.

   d. Replaces system names with system numbers.

7. Returns 0, indicating primary process.

In the backup process:

1. Inspects <flags>.<13:15> and calls the appropriate procedures, if any.

2. Determines that this is the backup of the named process pair.

3. Calls CHECKMONITOR. If CHECKMONITOR returns, this indicates that the primary process failed before it made a stack checkpoint.

   In this case, if <flags>.<12> = 0, the INITIALIZER calls ABEND; if <flags>.<12> = 1, the INITIALIZER returns -1, indicating the CHECKMONITOR failed.

Note: Normally CHECKMONITOR does not return; see section 5, "Checkpointing Facility".

The LASTADDR (last address) function procedure returns the 'G'[0] relative address of the last word in the application process's data area.

The LASTADDR function is invoked as follows:

```
<last address> := LASTADDR
                  --------

where

   <last address>, INT,

     is the 'G'[0] relative word address of the last word in
     the application process's data area.

example:

   highest^address := LASTADDR;
```

The LASTADDR function can be used to determine the number of memory pages allocated to a running application program:

```
num^pages := LASTADDR.<0:5> + 1;
```

A bit extraction is performed on the six high-order address bits returned from LASTADDR. One is added to that value (figure 4-1).

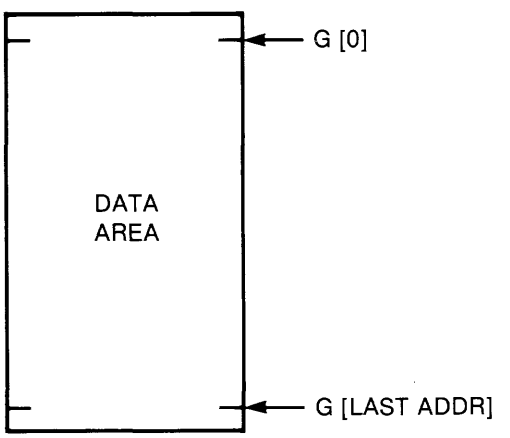


Figure 4-1. Last Address

The NUMIN function procedure converts ASCII representations of numbers, bases from 2 through 10, to signed integer values.

The NUMIN function is invoked as follows:

```
{ <next address> := } NUMIN ( <ascii number> , <signed result>
{ CALL             } ----- - -------------- - ---------------
                                , <base> , <status> )
                                - ------ - -------- -
```

where

<next address>, INT,

is the 'G'[0] relative string address of the first character in <ascii number> that was not used in the conversion.

<ascii number>, STRING:ref,

is an array containing the number to be converted to signed integer form. <ascii number> is of the form:

```
{ + } % <number> <nonnumeric>
{ - }   -------- -------------
```

where "%" means treat the number as an octal value regardless of the specified <base>.

<signed result>, INT:ref:1,

is a variable where NUMIN returns the result of the conversion.

<base>, INT:value,

specifies the number base of <ascii number>. Legitimate values are 2 through 10.

<status>, INT:ref:1,

is a variable where NUMIN returns a number that indicates the outcome of the conversion. The values for <status> are:

```
  1 = non-existent number (string does not start with "+",
      "-", "%", or numeric
  0 = valid conversion
 -1 = illegal integer (number cannot be represented in 15
      bits) or illegal syntax
```

→

```
example:

   @next^addr := NUMIN(in^buffer, number, 10, stat);
     or
   CALL NUMIN(in^buffer, number, 10, status);
```

CONSIDERATIONS

- Number conversion stops on the first ASCII numerical character representing a value greater than <base>-1 or non-numerical ASCII character.
- Base-10 numerical values must be in the range of {-32768:32767}. Numerical values in other number bases will be accepted if they can be represented in 16 bits. Note that the magnitude is computed first, then the value is possibly negated (e.g., %177777 = -%1).

Examples of NUMIN:

The value of NUMIN can be used to determine the number of characters converted:

```
STRING number [0:9] := "12345     ";
INT result, status, .next^char
```

Then NUMIN is invoked:

```
@next^char := NUMIN( number , result , 10 , status );
```

After NUMIN executes, the pointer variable "next^char" contains the address of "number[5]" (the sixth element).

Then subtracting

```
num^converted := @next^char '-' @number;
```

provides the number of characters used in the conversion (i.e., five).

An alternate way of doing the same:

```
num^converted := NUMIN(number,result,10,status) '-' @number;
```

Another example, this time showing a string containing an ASCII number greater than the base being converted:

```
STRING number[0:5] := "%19234";
```

Then NUMIN is invoked:

```
@next^char := NUMIN(number, result, 8, status);
```

The only character converted to its octal representation is "1". At completion, the pointer variable "next^char" points to the character "9".

The NUMOUT procedure converts unsigned integer values to their ASCII equivalents using any number base from 2 through 10. The result is returned right-justified in an array, filled with leading zeroes.

The call to the NUMOUT procedure is:

```
CALL NUMOUT ( <ascii result> , <unsigned integer> , <base>
---- ------ - -------------- - ------------------ - ------
              , <width> )
              - ------- -
```

where

<ascii result>, STRING:ref:*,

is the array where NUMOUT returns the converted value. The ASCII representation is returned right-justified in <ascii result> [<width> - 1], filled with leading zeros.

<unsigned integer>, INT:value,

is the value to be converted.

<base>, INT:value,

is the number base desired for the resultant conversion.

<width>, INT:value,

is the maximum number of characters permitted in <ascii result>. Characters may be truncated on the left side.

example:

```
CALL NUMOUT(out^buffer, err^num, 8, 5);
```

For example, an application wants to convert an INT value to its base-10 ASCII equivalent:

```
STRING array[0:5];
INT variable := 2768;
LITERAL base = 10, width = 6;

    .
    .
CALL NUMOUT(array, variable, base, width);
```

After NUMOUT executes, "array" contains:

"002768"

Another example, using the same number but converting to base 8:

```
    .
    .
CALL NUMOUT(array, variable, 8, width);
    .
```

After NUMOUT executes, "array" contains:

"005320"

A final example, using the same number and converting to base 10 but with a "width" of 3:

```
    .
CALL NUMOUT( array, variable, 10, 3);
    .
```

After NUMOUT executes, "array" contains:

"768"

The result is truncated to three characters; the three leftmost characters are lost.

The SHIFTSTRING procedure upshifts or downshifts all alphabetic characters in a string. Non-alphabetic characters remain unchanged.

The call to the SHIFTSTRING procedure is:

```
CALL SHIFTSTRING ( <string> , <count> , <casebit> )
---- ----------- - -------- - ------- - --------- -

where

   <string>, STRING:ref,

     is the character string to be shifted.

   <count>, INT,

     is the length of the string in bytes.

   <casebit>, INT,

     indicates whether to upshift or downshift the string.  If
     this parameter is even, the procedure upshifts, making all
     alphabetic characters upper-case; if it is odd, the procedure
     downshifts, making all alphabetic characters lower-case.

example:

   CALL SHIFTSTRING ( command, command^len, 0 );    ! upshift
```

The TIME procedure provides the current date and time in integer form.

The call to the TIME procedure is:

```
CALL TIME ( <date and time> )
---- ---- - --------------- -

where

   <date and time>, INT:ref:7,

     is an array where TIME returns the current date and time in
     the following form:

       <date and time>[0] = year     (1978, 1979, ... )
       <date and time>[1] = month    (1-12)
       <date and time>[2] = day      (1-31)
       <date and time>[3] = hour     (0-23)
       <date and time>[4] = minute   (0-59)
       <date and time>[5] = second   (0-59)
       <date and time>[6] = .01 sec  (0-99)

example:

   CALL TIME(time^array);
```

The TIMESTAMP procedure provides the internal form of the CPU interval clock where the application is running.

The call to the TIMESTAMP procedure is:

```
CALL TIMESTAMP ( <interval clock> )
---- --------- - ---------------- -

where

   <interval clock>, INT:ref:3,

     is an array where TIMESTAMP returns the current value of the
     interval clock.  A processor's interval clock is incremented
     every .01 second.  <interval clock> is returned in the
     following form:

       <interval clock>[0] = most significant word

       <interval clock>[1]

       <interval clock>[2] = least significant word

example:

   CALL TIMESTAMP(clock);
```

TOSVERSION Procedure

The TOSVERSION function procedure provides an identifying letter and number indicating which version of the GUARDIAN operating system is running.

The TOSVERSION function is invoked as follows:

```
<version> := TOSVERSION
             ----------

where

   <version>, INT,

     is returned a value of the form

          <0:7>   upper-case ASCII letter indicating system
                  level:
                  "A" = T.O.S.
                  "B" = GUARDIAN
                  "C" = GUARDIAN / 1.1
                  "D" = GUARDIAN / EXPAND
                  "E" = GUARDIAN / EXPAND / TMF
                  "K" = GUARDIAN, NonStop II system
          <8:15>  revision number of system, in binary
```

SECTION 5

CHECKPOINTING FACILITY

To aid in the development of NonStop programs, the checkpointing facility, which is an integral part of the GUARDIAN operating system, is provided.

The following topics are covered in this section:

- Overview of Checkpointing Procedures
- Overview of NonStop Programs
- Overview of Checkpointing

OVERVIEW OF CHECKPOINTING PROCEDURES

The checkpointing facility consists of a set of procedures which are used to

- Transfer control to the backup in case of failure of the primary process or its processor module:

  CHECKMONITOR (backup process)

- Open and close a process pair's files:

  OPEN and CLOSE (primary process)
  CHECKOPEN and CHECKCLOSE (primary process)
  CHECKMONITOR (backup process)

- Checkpoint a primary's execution state to its backup process:

  CHECKPOINT or CHECKPOINTMANY (primary process)
  CHECKMONITOR (backup process)

  The following types of information are checkpointed:

  - the primary process's data stack (defines a "restart" point)
  - individual arrays (e.g, file buffers) in the application process's data area

  - for disc files, the file's "synchronization block"

- Transfer control to the backup process so that the system load is redistributed:

  CHECKSWITCH (primary process)
  CHECKMONITOR (backup process)

- Monitor the operational state of one or more processor modules:

  MONITORCPUS (primary and backup processes)

- Obtain the count and operational states of processor modules:

  PROCESSORSTATUS

OVERVIEW OF NonStop PROGRAMS

The actions of the primary and backup processes of a NonStop program are shown in figure 5-1.

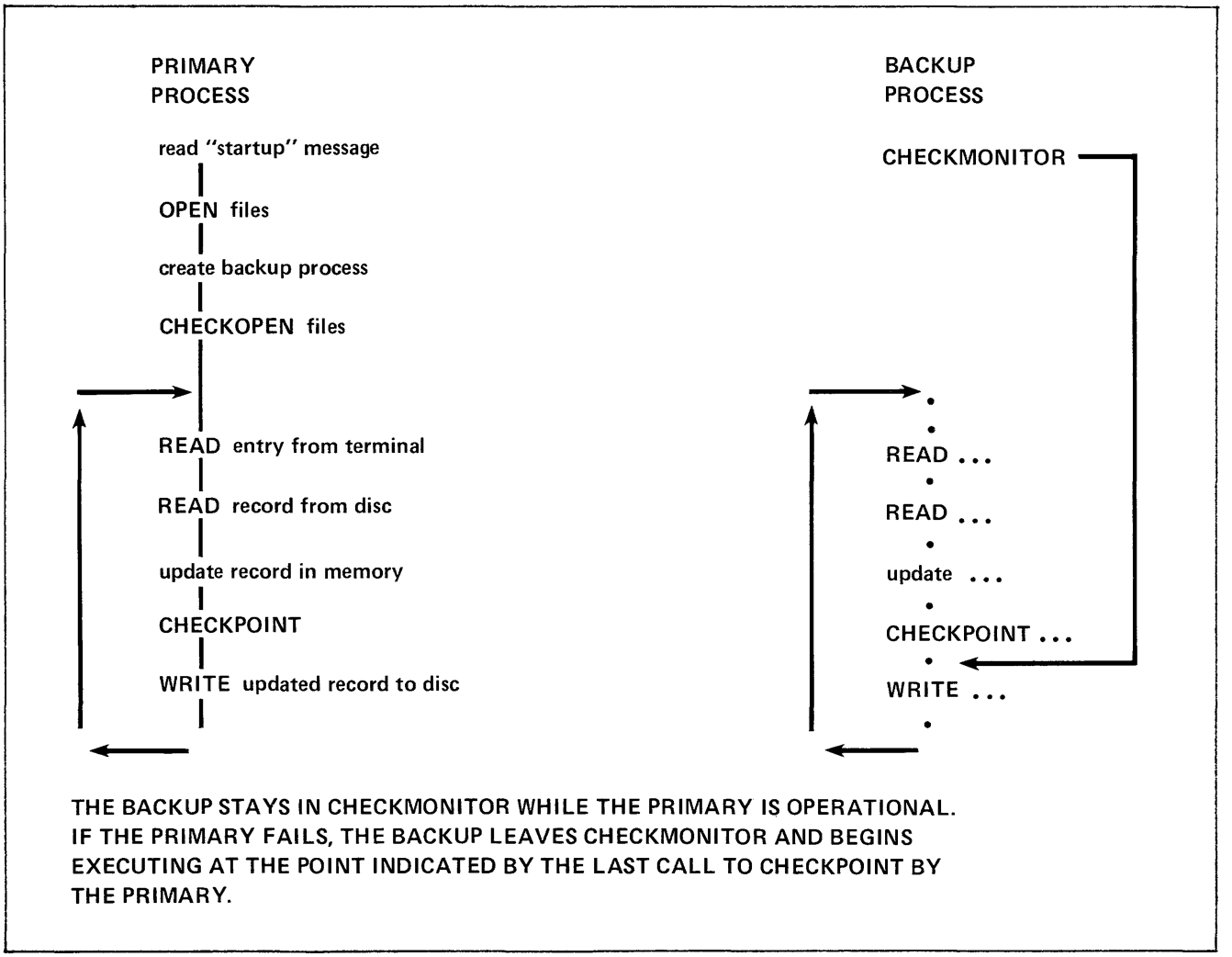


Figure 5-1. A NonStop Program

Basically, the following actions take place when a NonStop program runs:

1. First, the program is given a process name at run time. This permits the new process (and eventually its backup) to run as a named process pair and therefore take advantage of facilities of the Process-Pair Directory (PPD) (see "Process-Pair Directory" in section 3, "Process Control"). (Note: An alternate, more primitive method of setting up a NonStop program is to use two non-named processes and have each call the STEPMOM procedure to "adopt" the other.)

2. The new process, which typically is designated the primary process, reads the startup message from its creator (e.g., a Command Interpreter).

3. The primary process opens any files required for its execution.

4. The primary process then creates the backup process in another processor module. The backup process is given the same process name as the primary.

5. The backup process, at the beginning of its execution, calls the CHECKMONITOR procedure. This is as far as the backup executes unless a failure of the primary process occurs.

6. The primary process opens the same files for the backup process via calls to CHECKOPEN. This permits files to be open by the pair in a manner that permits both members of the pair to have a file open while retaining the ability to exclude other processes from accessing a file. For disc files open in this manner, a record or file lock by the primary is also an equivalent lock by the backup.

7. The primary process then begins executing its main processing loop. At critical points through the execution loop, typically before writes to disc files, the primary calls CHECKPOINT to send part of its environment and pertinent file control information to the backup process. Typically, a program contains several calls to CHECKPOINT; each call checkpoints only a portion of the primary process's environment. Calls to CHECKPOINT that checkpoint the data stack define restart points for the backup process.

8. If the primary process fails, the backup begins executing at the restart point indicated by the latest call to CHECKPOINT that checkpointed the data stack. The backup process is now considered to be the primary process.

9. If the reason for the primary process failure was a processor module failure (i.e., cpu down), the new primary process creates a backup process when the failed processor module is repaired and brought back online. This new backup process is then ready to take over if the primary process fails. (This is the normally recommended procedure; an alternative action is to create a backup process immediately in another cpu.)

OVERVIEW OF CHECKPOINTING

The following types of information can be checkpointed:

- the process's data stack

  The data stack, in this context, is considered to be the area from an address specified in the call to CHECKPOINT (usually the address of the last global variable) through the current top-of-stack location (i.e., the word pointed to by the current setting of the S register). This area contains the local data storage for all currently active procedures and their stack markers.

- individual blocks of data in the data area

  These are usually file buffers, but may be any data desired.

- disc file "sync blocks"

  A "sync block" contains control information about the current state of a disc file (e.g., current value of the file pointers).

When a call to CHECKPOINT is made by the primary process, a message containing the information to be checkpointed is formatted and sent to the backup process in the form of an interprocess message. The message is received and processed by the CHECKMONITOR procedure in the backup process.

Data Stack

The purpose of checkpointing the data stack is to provide a restart point for the backup process. This is possible because the stack markers in the data stack define the executing environment of the primary process at the time of the call to CHECKPOINT, and because the primary's data stack is duplicated in the backup. If the primary process fails, CHECKMONITOR simply returns through the stack marker for the latest call to CHECKPOINT. In this manner, the backup begins executing following the latest call to CHECKPOINT.

Data Buffers

The purpose of checkpointing data buffers is to preserve the state of the process so that the backup can continue processing. Typically, data buffer checkpointing occurs just before writing to a disc file; the data about to be written is checkpointed. Careful selection of data buffers (and corresponding file sync information, discussed in the following paragraphs) to checkpoint can increase the efficiency of a NonStop program. An example of data buffer checkpointing is an entry received from a terminal; the data buffer is checkpointed to minimize the possibility that the operator would have to reenter data. Note that data buffers residing in the data stack are checkpointed when the stack is checkpointed.

Sync Blocks

The purpose of checkpointing the sync block is twofold:

1. To ensure that no write operation is duplicated when a backup takes over from its primary

2. To pass the current values of file pointers to the file system on the backup side

When a checkpoint of the sync block occurs, the information in the sync block is passed to the file system by CHECKMONITOR.

The reason for preventing duplicate operations is illustrated in the following sequence:

A primary completes the following write operation successfully, but fails before a subsequent checkpoint to its backup -

```
RESTART POINT ------>(C)   CHECKPOINT POSITION AND DATA
                      |
                     x  POSITION(fl,-1D); ! position to eof
                     x  WRITE(fl,fl^buffer);
                      |
            *** FAILURE OF PRIMARY ***
```

On the takeover from the primary, the backup reexecutes the operations just completed by the primary. If the WRITE were performed as requested, it would duplicate the record, but at the new end-of-file location.

So that no write operation already performed by the primary will be duplicated by the backup process, the <sync depth> parameter of OPEN must be specified as a value greater than zero when opening the file. For a file open in this manner, a sync ID in the sync block is used by the file system in the event of a primary process failure to identify the operation about to be performed by the backup. If the backup requests an operation already completed by the primary, the file system, through use of the sync ID, recognizes this condition. Then, instead of performing the requested operation, the file system returns the completion status of the operation to the backup (the completion status was saved by the file system when the primary performed the operation). However, if the requested operation had not been performed, it is performed and the completion status is returned to the backup. The course of action that is taken by the file system is completely invisible to the backup process.

The file system has the capability to save the completion status of the latest 15 operations with a file and to relate those completions with up to 15 operations requested by a backup process upon takeover from a failed primary process. The maximum number of completion statuses that the file system is to save is specified in the <sync depth> parameter to OPEN. The sync depth value is typically the same as the maximum number of write operations to a file without an intervening checkpoint of the file's sync block. In most cases, the sync depth value is 1. It cannot exceed 15.

The checkpointing procedures are:

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

CHECKMONITOR — is called by a backup process to monitor the operability of its primary process. CHECKMONITOR performs two functions: 1) it performs the operations required when CHECKOPEN, CHECKPOINT, or CHECKCLOSE is called in the primary process, and 2) it returns control to the appropriate point in the backup process in the event that a failure of the primary process or processor occurs or if the primary calls CHECKSWITCH

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

CHECKPOINT — is called by a primary process to checkpoint its data stack, local file buffers, and/or file synchronization information to its backup process. The data stack and any combination of up to 13 data blocks or file sync blocks can be checkpointed in a single call

CHECKPOINTMANY — has the same function as CHECKPOINT, except that it allows an unlimited number of data blocks and file sync blocks to be checkpointed in a single call

CHECKSWITCH — is called by a primary process to switch control to its backup process. A call to CHECKSWITCH is an implicit call to CHECKMONITOR so that the primary process becomes the backup process

MONITORCPUS — instructs the GUARDIAN operating system to notify the caller if the operating state of a designated processor module changes from an operable to a non-operable state or from a non-operable to an operable state

PROCESSORSTATUS — returns a count of the number of processors in the system and the up-down state of each processor

Note: The following procedures are called implicitly by the "CHECK" procedures, and therefore are not normally called explicitly. However, they can be used by application programmers when writing application-dependent failure recovery techniques:

GETSYNCINFO — is called by a primary process to acquire a disc file's sync information so that it can be can be sent to its backup process

RESETSYNC	is called by a backup process, following a takeover from its primary, to clear a disc file's sync information on the backup side. RESETSYNC is called prior to reexecuting disc operations when the backup wants the operation to occur regardless of whether or not the operation has already been performed by the primary. RESETSYNC is also called to resynchronize any open files whose file sync blocks were not checkpointed after the most recent stack checkpoint
SETSYNCINFO	is called by a backup process, following a takeover from its primary, to set a disc file's sync information on the backup side. SETSYNCINFO is called prior to reexecuting disc operations that may have just been performed by the primary so that already-completed operations will not be repeated

CONSIDERATIONS

- If a file is open with no-wait i/o specified, the following calls are rejected with a file management error 27 if there are any outstanding (i.e., uncompleted) operations pending:

  GETSYNCINFO
  RESETSYNC
  SETSYNCINFO

- A call to

  CHECKCLOSE,
  CHECKOPEN,
  CHECKPOINT,
  CHECKPOINTMANY, or
  CHECKSWITCH

  causes an interprocess message to be sent to the process indicated by the "creator process ID" in the caller's Process Control Block. The creator process ID is automatically set to the process ID of the backup process at process creation if the primary/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 one of these procedures. (The interprocess message is received and processed by CHECKMONITOR in the backup process.)

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 (i.e., 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 management CLOSE procedure for the designated file.

The call to the CHECKCLOSE procedure is:

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

where

   <file number>, INT:value,

     identifies the file to be closed in the backup process.

   <tape disposition>, INT:value,

     if present, specifies mag tape disposition:

       where

       <tape disposition>.<13:15>

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

     if omitted, 0 is used.

condition code settings (from the CLOSE in the backup 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) is not returned by CHECKCLOSE.

example:

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

CONSIDERATIONS

- The condition code returned from CHECKCLOSE indicates the outcome of the CLOSE in the backup process.
- See the considerations for "CLOSE" in the "File Management Procedures" section.

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 call to the CHECKMONITOR procedure is:

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

where

<status>, INT,

is returned a status word of the following form:

<0:7> = 2, <8:15> = 0 primary stopped
1 primary abended
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 via a call to CHECKPOINT.

example:

```
CASE CHECKMONITOR.<8:15> OF
  BEGIN
    .
    .
  END;
```

CONSIDERATIONS

- If the process pair is not named (i.e., not in the PPD), the STEPMOM procedure must be called prior to the call to CHECKMONITOR and before the primary process makes its first call to CHECKPOINT.
- While CHECKMONITOR executes, its local data area consists of approximately 500 words starting at

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

- that is, to 500 words below the last available location in the application process's data stack. This region is used by CHECKMONITOR to call other operating system procedures. If the primary attempts to checkpoint its data area in this region, then an "illegal parameter" error is returned to the primary process from CHECKPOINT.

  If this failure occurs, then the number of data pages to be allotted the process should be increased via the "?DATAPAGES" TAL compiler command. (This method of increasing data area size should be used, rather than increasing the data area at run time via the Command Interpreter MEM parameter, to avoid creating a backup with a different data area size than its primary.)

- The specific action of CHECKMONITOR for an action by the primary process is given in table 5-1.

Table 5-1. Action of CHECKMONITOR

Primary	Backup (CHECKMONITOR)
No action	At beginning of CHECKMONITOR execution, the current state of cpu monitoring for the caller is saved (i.e., the current MONITORCPUS <cpu mask>), then MONITORCPUS is called, specifying only the primary's processor module.
CHECKOPEN	OPEN is called for the designated file.
CHECKPOINT	If all or a portion of the primary's data stack was checkpointed, the data is moved into the corresponding location in the backup's data stack. If a local data buffer was checkpointed by name, the data is moved into the appropriate location in the backup's data area. If file synchronization information was checkpointed, SETSYNCINFO is called for the designated file.
CHECKCLOSE	CLOSE is called for the designated file.
CHECKSWITCH	First CHECKMONITOR calls RESETSYNC for any file whose synchronization information the primary did not checkpoint in its preceding call to CHECKPOINT. CPU monitoring is returned to the state that was in effect before CHECKMONITOR was called. Control is then returned to the point in the backup process as indicated by the latest call to CHECKPOINT in the primary process. If the primary has not previously checkpointed its stack in a call to CHECKPOINT, control is returned to the instruction following the call to CHECKMONITOR.
Process Failure	(STOP or ABEND system message received for primary). First CHECKMONITOR calls RESETSYNC for any file whose synchronization information the primary did not checkpoint in its preceding call to CHECKPOINT. CPU monitoring is returned to the state that was in effect before CHECKMONITOR was called. Control is then returned to the point in the backup process as indicated by the latest call to CHECKPOINT in the primary process. If the primary has not previously checkpointed its stack in a call to CHECKPOINT, control is returned to the instruction following the call to CHECKMONITOR.

Table 5-1. Action of CHECKMONITOR (cont'd)

Primary	Backup (CHECKMONITOR)
Processor Failure	(Processor Failure system message received for primary's processor module). First CHECKMONITOR calls RESETSYNC for any file whose synchronization information the primary did not checkpoint in its preceding call to CHECKPOINT. CPU monitoring is returned to the state that was in effect before CHECKMONITOR was called. Control is then returned to the point in the backup process as indicated by the latest call to CHECKPOINT in the primary process. If the primary has not previously checkpointed its stack in a call to CHECKPOINT, control is returned to the instruction following the call to CHECKMONITOR.

The CHECKOPEN procedure is called by a primary process to open a designated file for its backup process. The file must first be opened by the primary process. The backup process must be in the "monitor" state (i.e., 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 management OPEN procedure for the designated file.

The call to the CHECKOPEN procedure is:

```
CALL CHECKOPEN ( <file name> , <file number>
---- --------- - ----------- - -------------
                , <flags>
                -
                , <sync or receive depth>
                -
                , <sequential block buffer>
                -
                , <buffer length>
                -
                , <back error> )
                - ------------ -
```

where

the following parameters must be passed the same values as those passed for the corresponding parameters in the call to OPEN for this file:

CHECKOPEN parameter	corresponding OPEN parameter
<file name>, INT:ref	= <file name>
<file number>, INT:value	= <file number>
<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>

The following parameter is required:

<back error>, INT:ref:1,

>= 0, is the file management 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.

  Condition code settings (from the OPEN in the backup process):

     <  (CCL) indicates that the OPEN failed.  The file management
              error number is returned in <back error>.
     =  (CCE) indicates that the file opened successfully.
     >  (CCG) indicates that the OPEN was successful, but an
              exceptional condition was detected.  The file
              management error number is returned in <back error>.

  example:

     CALL OPEN ( filename, filenum );
     IF < THEN ....                    ! OPEN failed for primary.
       .
     CALL CHECKOPEN ( filename, filenum,,,,, error );
     IF < THEN ...                     ! OPEN failed for backup.
```

CONSIDERATIONS

- The condition code returned from CHECKOPEN indicates the outcome of the OPEN in the backup process.
- See the considerations for OPEN in the appropriate programming manual (i.e., GUARDIAN, ENSCRIBE, or ENVOY).
- If an "unable to communicate with backup" error occurs (i.e., <back error> = -1), this normally indicates either that the backup process does not exist or that a system resource problem exists. If a system resource problem is indicated, then either the open request message to the backup is unduly large or the SHORTPOOL size in the processor module where the error occurs is too small.
- <back error> = 17 is returned if the file is not open by the primary process or the parameters supplied to CHECKOPEN do not match the parameters supplied when the primary process opened the file.
- If a process file is opened in a no-wait manner (<flag>.<8> = 1), that file is CHECKOPENed as no-wait. Errors detected in parameter specification and system data space allocation are returned by CHECKOPEN in <backerr>, and the operation is considered complete. If no error is returned in <backerr>, the operation must be completed by a call to AWAITIO in the primary process. The tag value returned by AWAITIO is -29D if the <tag> parameter is specified; the returned count and buffer address are undefined. If

CCL is returned by AWAITIO, the file has been automatically checkclosed by the checkpointing facility. For a non-process file, or a process file that was wait-opened, bit 8 is reset internally to 0 and ignored. The user can call AWAITIO to complete CHECKOPENs which he was required to complete for the primary open of the file, by calling AWAITIO.

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

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

The call to the CHECKPOINT procedure is:

```
{ <status> := } CHECKPOINT ( <stack base>
{ CALL       } ----------- -
                           , <buffer 1>   , <count 1>
                           , <buffer 2>   , <count 2>
                                .              .
                                .              .
                                .              .
                           , <buffer 13> , <count 13> )
                                                      -
```

where

<status>, INT,

is returned a status word of the following form:

```
<0:7> = 0, no error
        1, no backup or unable to communicate with backup
           <8:15> = file management error number
        2, takeover from primary, then
           <8:15> = 0, primary stopped
                    1, primary abended
                    2, primary's processor failed
                    3, primary called CHECKSWITCH
        3, illegal parameter, then
           <8:15> = number of parameter in error
                    (leftmost position = 1)
```

<stack base>, INT:ref,

if present, 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>, INT:ref,

if present, 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 <buffer n> is omitted, <count n> is treated as a <file number>, and that file's file synchronization block is checkpointed.

<count n>, 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 <file number> of a file whose synchronization block is to be checkpointed.

example:

```
@p := 0; ! beginning of global area
stat := CHECKPOINT (p,,fnum^a,,fnum^b);
```

CONSIDERATIONS

- If an "unable to communicate with backup" error occurs, this normally indicates either that the backup process does not exist or that a system resource problem exists. If a system resource problem is indicated, then 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.

- If an attempt is made to checkpoint the data area in the region used by CHECKMONITOR in the backup process, then an "illegal parameter" error is returned. See the "Considerations" for CHECKMONITOR for the recovery procedure.

- If a file's sync information is checkpointed, the call to the 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 the file. If file sync information checkpointing is performed, FILEINFO returns the status of the call to GETSYNCINFO (usually, <error> = 0).

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 CHECKPOINTMANY procedure provides for checkpointing the process's data stack and any number of separate data blocks and file synchronization blocks, limited by system limits on the size of the resulting message.

The call to the CHECKPOINTMANY procedure is:

```
{ <status> := } CHECKPOINTMANY ( <stack base>
{ CALL       } -------------- -
                                , <descriptors> )
                                                -
```

where

<status>, INT,

is returned a status word of the following form:

```
<0:7> = 0, no error
        1, no backup or unable to communicate with backup
           <8:15> = file management error number
        2, takeover from primary, then
           <8:15> = 0, primary stopped
                    1, primary abended
                    2, primary's processor failed
                    3, primary called CHECKSWITCH
        3, illegal parameter, then
           <8:15> = (see "Considerations" below)
```

<stack base>, INT:ref,

if present, 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>, INT:ref,

if present, is an array which describes the items (data blocks and/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.

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>[1st] = -1
<descriptors pair>[2nd] = <file number>
```

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 of the pair is the length, in words, of the data block:

```
<descriptors pair>[1st] = <buffer>
<descriptors pair>[2nd] = <count>
```

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

```
1 + 2 * <descriptors[0]>.
```

example:

```
desc[0] := 2;          ! count of items
desc[1] := -1;         ! sync item:
desc[2] := fnum^a;     !    file number
desc[3] := @buffer;    ! data item:  word address
desc[4] := 512;        !    number of words
stat:= CHECKPOINTMANY( stk^base, desc );
   ! this is equivalent to:
   ! stat := CHECKPOINT( stk^base,,fnum^a,buffer,512 );
```

CONSIDERATIONS

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

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

  Following word 0, <descriptor> consists of pairs of words. If the pair describes a file sync block (first word of pair = -1, second word = file number) then <descriptor>[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 <descriptor>[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 <descriptor>[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 equal to 2 * descriptor[0].

- If an attempt is made to checkpoint the data area used by CHECKPOINTMANY for system-oriented stack maintenance, then an "illegal parameter" error is returned.

- Also see the "Considerations" for "CHECKPOINT".

The CHECKSWITCH procedure is called by a primary process to cause the duties of the process pair to be interchanged. CHECKSWITCH is intended to be used 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 begin processing from the latest call to CHECKPOINT. The call to CHECKSWITCH contains an implicit call to CHECKMONITOR, so that the caller becomes that backup and monitors the execution state of the new primary. The backup process must be in the "monitor" state (i.e., in a call to CHECKMONITOR) for the CHECKSWITCH to be successful.

The call to the CHECKSWITCH procedure is:

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

where

   <status>, INT,

     on return, returns a status word of the following form:

       <0:7> = 1, could not communicate with backup, then
                  <8:15> = file management error number
       <0:7> = 2, <8:15> = 0  primary stopped
                           1  primary abended
                           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 was checkpointed.

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

example:

   stat := CHECKSWITCH;
```

CONSIDERATIONS

- See the CHECKMONITOR procedure for the action of CHECKSWITCH following the takeover by the backup process.

GETSYNCINFO Procedure (disc and process files)

Note: Typically, GETSYNCINFO is not called directly by application programs. Instead, it is called indirectly by CHECKPOINT.

The GETSYNCINFO procedure is called by the primary process of a primary/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.

The call to the GETSYNCINFO procedure is:

```
CALL GETSYNCINFO ( <file number> , <sync block>
                 , <sync block size> )
```

where

<file number>, INT:value,

identifies the file whose sync block is to be obtained.

<sync block>, INT:ref:*,

is returned 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, =

  7 + (longest alt key len + pri key len + 1) / 2

<sync block size>, INT:ref:1,

is returned 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.

example:

```
CALL GETSYNCINFO ( file^num, sync^id );
IF < THEN .....;          ! error
```

The MONITORCPUS procedure instructs the GUARDIAN operating system to notify the application process if a designated processor module fails (a failure being indicated to the operating system by the non-receipt of an operating system "I'm alive" message) or returns from a failed to an operable state (i.e., reloaded by means of a Command Interpreter RELOAD command). The calling application process is notified by a means of a system message read via the $RECEIVE file.

The call to the MONITORCPUS procedure is:

```
CALL MONITORCPUS ( <cpu mask> )
---- ----------- - ---------- -

where

   <cpu mask>, INT:value,

     has a bit set to "1"corresponding to each processor module
     to be monitored:

       <cpu mask>.<0>  = processor module 0
       <cpu mask>.<1>  = processor module 1
          .
          .
       <cpu mask>.<15> = processor module 15

     <cpu mask> = 0 means no notification occurs.

example:

   CALL MONITORCPUS ( %140000 ); ! cpu's 0 & 1
```

The system messages associated with MONITORCPUS, in word elements, are:

- CPU Down message. There are two forms of the CPU Down message:

```
<sysmsg>            = - 2
<sysmsg>[1]         = <cpu>
```

  This form is received if a failure occurs with a processor module being monitored. Monitoring for specific processor modules is requested by a call to the process control MONITORCPUS procedure.

and

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

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

- CPU Up message:

```
<sysmsg>            = - 3
<sysmsg>[1]         = <cpu>
```

This message is received if a reload occurs with a processor module being monitored.

The PROCESSORSTATUS procedure is used to obtain a count of the number of processor modules in a system and their operational states.

The call to the PROCESSORSTATUS procedure is:

```
<processor status> := PROCESSORSTATUS
                      ---------------

where

   <process status>, INT(32),

     is returned two words indicating the count and states of
     processor modules.

     The most significant word is the count of processor modules.

     The least significant word is a bit mask indicating the
     operational state of each processor module:

       <ls word>.<0>  = processor module 0
       <ls word>.<1>  = processor module 1
          .
          .
       <ls word>.<15> = processor module 15

       A "1" indicates that the corresponding processor module is
       up (i.e., operational).  A "0" indicates that the
       corresponding processor module is down or does not exist.

example:

   INT(32) cpu^info;
   INT num^cpus  = cpu^info,
       cpu^state = cpu^info + 1;

   cpu^info := PROCESSORSTATUS;
```

RESETSYNC Procedure (disc and process files)

Note: Typically, RESETSYNC is not called directly by application programs. Instead, it is called indirectly by CHECKMONITOR.

The RESETSYNC procedure is used by the backup process of a primary/backup process pair after a failure of the primary process when a different series of operations will be performed than those of the primary before its failure. The RESETSYNC procedure clears a paired access file's synchronization block so that the operations to be performed by the backup are not erroneously related to the operations just completed by the primary process.

RESETSYNC is also called to resynchronize any open files whose file sync blocks were not checkpointed after the most recent stack checkpoint.

The call to the RESETSYNC procedure is:

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

where

   <file number>, INT:value,

     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 was successful.
   > (CCG) indicates that the file is not a disc file.

example:

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

Note: Typically, SETSYNCINFO is not called directly by application programs. Instead, it is called indirectly by CHECKMONITOR.

The SETSYNCINFO procedure is used by the backup process of a primary/backup process pair after a failure of the primary process. The SETSYNCINFO procedure passes a paired accessed file'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 synchronization block ensures that operations that may have been completed by the primary before its failure are not duplicated by the backup.)

The call to the SETSYNCINFO procedure is:

```
CALL SETSYNCINFO ( <file number> , <sync block> )
---- ----------- - ------------- - ------------ -

where

   <file number>, INT:value,

     identifies the file whose synchronization block is being
     passed.

   <sync block>, 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 was successful.
   > (CCG) indicates that the file is not a disc file.

example:

   CALL SETSYNCINFO ( file^num, syncid );
   IF < THEN .....;           ! error
```

This section describes

- The general structure of a NonStop program
- Considerations for opening files
- General considerations for checkpointing
  - Example of where checkpoints should occur
  - Sync Blocks for disc files
  - Checkpointing multiple disc updates
  - Considerations for no-wait i/o
- The action to take when a system message is read
- The action to take when the backup takes over
- How to open a file during processing
- How a process pair should create a descendant process [pair]

NonStop PROGRAM STRUCTURE

The general structure of a typical NonStop program is

- a process startup (beginning of program) phase, and
- a main processing loop.

Process Startup for Named Process Pairs

The use of named process pairs for NonStop programming is considered to be the usual case. Non-named process pairs are used only in special cases.

The process startup code is executed by both the primary and backup processes following their creation.

The general steps involved in process startup are:

1. Save the stack base address for checkpointing.
2. Call ARMTRAP so process will abend if trap occurs.
3. Determine if the process is the primary or backup -

```
     If primary then
       begin
4.       open $RECEIVE (no-wait) and, optionally, read startup message
5.       open files
6.       monitor the backup cpu
7.       create backup process:
           if created then
             begin
8.             open files in backup process
9.             use Awaitio to complete nowait opens in backup process
10.            checkpoint environment to backup
             end
       end
```

11. else ! backup ! monitor the primary.
12. Initiate a read on $RECEIVE to check for backup stopped or processor up/down messages.

After performing these steps, execute the main processing loop.

1. SAVE THE STACK BASE ADDRESS: This is necessary for subsequent checkpointing of the data stack. The stack base address should be kept in a global variable. The stack base address is that of the first local variable of the main procedure:

```
INT .stackbase; ! global pointer variable.

PROC m MAIN;
  BEGIN
    INT .ppdentry[0:8],   ! first local variable in MAIN.
          .
          .
        base = 'L' + 1;   ! address equivalence.

    @stackbase := @base;  ! saves the address.
```

2. CALL ARMTRAP: The ARMTRAP procedure should be called to handle any trap that may occur. The simplest method of using ARMTRAP is

```
    CALL ARMTRAP ( 0, -1 );
```

This causes the process to abend if a trap occurs.

The process may wish to analyze the reason for the trap. If this is desired, refer to section 6, "Traps and Trap Handling".

Note: During the program debug phase, it is usually desirable to omit the call to ARMTRAP. Then, if the process traps, DEBUG will be called.

3. DETERMINE IF PRIMARY OR BACKUP: One way to determine if a process is a primary or its backup is to look at its entry in the Process-Pair Directory (PPD):

```
    INT .ppdentry[0:8];

    CALL GETCRTPID ( MYPID, ppdentry );
    CALL LOOKUPPROCESSNAME ( ppdentry );
    IF < THEN CALL ABEND; ! no entry.
```

This returns the PPD entry for this process. If LOOKUPPROCESSNAME fails, either the process does not have a name or the system cannot access the PPD. In either case, a serious problem exists.

```
    IF NOT ppdentry[4] THEN ! i'm the primary
      BEGIN
        .
```

The fact that ppdentry[4] (i.e., <cpu,pin 2>) = 0 indicates that no backup has ever been created. Therefore, this process must be the primary.

The following actions are taken by the primary process:

4. OPEN $RECEIVE: The $RECEIVE file should be opened with no-wait i/o specified. No-wait i/o is specified so that a read on $RECEIVE can be continually outstanding. This is desirable so that the "check for completion" form of AWAITIO (i.e., <time limit> = 0D) can be used to check for system messages or so that system messages can be read when waiting for completions on other files.

```
INT .receive[0:11] := ["$RECEIVE", 8 * ["  "]], ! global
     rfnum,                                     ! variables.

 .
CALL OPEN ( receive, rfnum, 1 );
IF < THEN CALL ABEND;
```

Next, if a startup message is expected (e.g., Command Interpreter parameter message), it should be read:

```
CALL READ ( rfnum, buf, count );
IF <> THEN CALL ABEND;
CALL AWAITIO ( rfnum,, countread );
IF <> THEN CALL ABEND;
```

At this point, a check should be made to determine if the message is a valid startup message (i.e., first word of the message = -1).

5. OPEN PRIMARY'S FILES: The files to be referenced by the process should be opened in the primary (see "File Open").

```
LITERAL                              ! global data declarations.
     flags1      = ...,              !
     sync^depth1 = ...,              !
     flags2      = ...,              !
     sync^depth2 = ...,              !
        .                            !
        .                            !
     flagsn      = ...,              !
     sync^depthn = ...;              !
                                     !
INT .fname1[0:11],                   !
     fnum1,                          !
    .fname2[0:11],                   !
     fnum2,                          !
       .                             !
       .                             !
    .fnumn[0:11],                    !
     fnumn;                          !
```

```
CALL OPEN ( fname1, fnum1, flags1, sync^depth1 );
IF < THEN .... ! see note.
CALL OPEN ( fname2, fnum2, flags2, sync^depth2 );
IF < THEN .... ! see note.
  .
  .
CALL OPEN ( fnamen, fnumn, flagsn, sync^depthn );
IF < THEN .... ! see note.
```

Note: The action that should be taken if a file open fails (i.e., error <> 0) is application-dependent. For example, the primary could abort itself. Or, if an invalid file name was received by the process, the terminal operator could be queried for a valid file name.

6. MONITOR THE BACKUP CPU: The MONITORCPUS procedure should be called for the backup process's processor module. This is necessary so that processor module failure/reload system messages will be sent to the primary process.

```
INT backup^cpu; ! backup cpu no.

  .
  ! monitor the backup cpu.
CALL MONITORCPUS ( %100000 '>>' backup^cpu );
  .
```

7. CREATE THE BACKUP PROCESS: Backup process creation is best accomplished by writing a procedure which performs the following functions:

- process creation
- opening of the backup's files
- checkpointing the primary's environment

The reason for including these functions in a procedure is that backup process creation may be necessary at several points during process execution. These are: during process startup, after a takeover by a backup following a failure of its primary, a failure of backup (ABEND), or a reload of the backup's processor module.

The following is an example of backup process creation:

```
PROC createbackup;
  BEGIN
```

Create the process:

```
INT .pfile[0:11],
    pname[0:3],
    backup^pid[0:3],
    error;
```

```
CALL PROGRAMFILENAME ( pfile );
```

returns the file name of the primary's program file.

```
CALL GETCRTPID ( MYPID, pname );
```

returns the process pair's name.

```
CALL NEWPROCESS ( pfile,, (LASTADDR'>>'10) '+' 1, backup^cpu,
                  backup^pid, error, pname );
```

creates the process. (For an explanation of the memory parameter, see the LASTADDR procedure in section 4.)

Open the files in the backup process (see "File Open" for considerations):

```
IF backup^pid THEN ! it was created.
  BEGIN

    backup^up := 1; ! global variable.

    ! $RECEIVE file.
    CALL CHECKOPEN ( receive, rfnum, 1,,,, error );
    IF <> THEN ... ! see note.
    CALL CHECKOPEN( fname1,fnum1,flags1,sync^depth1,,,error );
    IF <> THEN ... ! see note.
    CALL CHECKOPEN( fname2,fnum2,flags2,sync^depth2,,,error );
    IF <> THEN ... ! see note.
      .
      .
    CALL CHECKOPEN( fnamen,fnumn,flagsn,sync^depthn,,,error );
    IF <> THEN ... ! see note.
```

Note: The action that a primary should take if a file open in its backup fails (i.e., error <> 0) is application-dependent. For example, the primary could stop the backup, then abort itself. Or, the primary could stop the backup but continue processing without a backup. If the latter course of action is taken, however, the primary will receive a process STOP system message for the backup. Therefore, the primary should contain logic so that it does not re-create its backup.

Note: When a server is opened in a no-wait manner by a process pair, the OPEN and the CHECKOPEN must both have been completed without error by AWAITIO before the sync block is checkpointed. If this restriction is not obeyed, the CHECKOPEN is rejected with an error, and a takeover occurs, then the server may not recognize the backup as a valid opener. In this case, pending requests may be rejected with an error if retried without the backup process first opening the file on its own. When using no-wait opens, the primary process of a pair should create the backup in the following manner to ensure a valid takeover:

1. Create backup using the NEWPROCESS procedure.

2. CHECKOPEN all files.

3. Complete all no-wait CHECKOPENs by calls to AWAITIO.

4. Checkpoint the stack and Sync Blocks.

   If the primary process dies, the backup is now ready to continue processing. Normal processing can continue in parallel with step 3, which may take a while if one or more servers responds slowly.

Checkpoint the primary's data area to the backup process (this will include any startup message). If the data area is large, this may require multiple calls to CHECKPOINT due to operating system SHORTPOOL limitations:

```
CALL CHECKPOINT (, addr, count, ... );
  .
```

Checkpoint all files' sync information and the data stack in the same call:

```
  .
  .
! set restart point.
IF (status := CHECKPOINT(stackbase ,, fnum1,, fnum2,, ...
                         ,, fnumn )) THEN

   CALL analyze^checkpoint^status ( status );
```

"analyze^checkpoint^status" is a procedure which takes appropriate action for a checkpoint failure or takeover by backup. See "Takeover by Backup" for a description of the "analyze^checkpoint^status" procedure.

If multiple calls to CHECKPOINT are necessary, the data stack should be checkpointed last. This checkpoint is then a restart point if the primary should subsequently fail.

```
      END; ! open files
END; ! of createbackup
```

11. MONITOR THE PRIMARY: This is the action taken by the process if it is the backup. First, MONITORCPUS is called for the primary's processor module (this is done so that the primary's processor module will continue to be monitored if and when the backup takes over). The actual monitoring of the primary is accomplished by calling the CHECKMONITOR procedure:

```
    ELSE ! i'm the backup
      BEGIN
          ! save the primary's cpu num.
        backup^cpu := ppdentry[3].<0:7>;
          ! monitor the primary cpu.
        CALL MONITORCPUS ( %100000 '>>' backup^cpu );
        CALL CHECKMONITOR;
        CALL ABEND;
      END;
```

The backup process only returns from the call to CHECKMONITOR if the primary has not checkpointed its data stack. The primary checkpoints its stack for the first time at the end of creation of the backup process.

12. READ $RECEIVE: The primary should keep a read outstanding on $RECEIVE at all times. This is desirable so that process deletion and processor failure/reload system messages can be received.

```
    CALL READ ( rfnum, rbuf, count );
```

SUMMARY: The following is the example code for process startup:

```
INT  backup^cpu,                                    ! global data
    .stackbase, ! global pointer variable.          ! declarations.
    .receive[0:11] := ["$RECEIVE", 8 * ["  "]],     !
     rfnum,                                         !
     stop^count := 0,                               !
     backup^up := 0;                                !

LITERAL                                             !
     flags1      = ...,                             !
     sync^depth1 = ...,                             !
     flags2      = ...,                             !
     sync^depth2 = ...,                             !
        .                                           !
        .                                           !
     flagsn      = ...,                             !
     sync^depthn = ...;                             !
                                                    !
INT .fname1[0:11],                                  !
     fnum1,                                         !
    .fname2[0:11],                                  !
     fnum2,                                         !
       .                                            !
       .                                            !
    .fnumn[0:11],                                   !
     fnumn;                                         !

  .
PROC m MAIN;
  BEGIN
    INT .ppdentry[0:8],   ! first local variable in MAIN.
          .
```

```
        .
      base = 'L' + 1;  ! address equivalence.

  @stackbase := @base;  ! save the address.

    ! abort the process if a trap occurs.
  CALL ARMTRAP ( 0, -1 );

  CALL GETCRTPID ( MYPID, ppdentry );
  CALL LOOKUPPROCESSNAME ( ppdentry );
  IF < THEN CALL ABEND; ! no entry.

  IF NOT ppdentry[4] THEN ! i'm the primary
    BEGIN
        ! open $RECEIVE.
      CALL OPEN ( receive, rfnum, 1 );
      IF < THEN CALL ABEND;
        ! read the startup message.
      CALL READ ( rfnum, buf, count );
      IF <> THEN CALL ABEND;
      CALL AWAITIO ( rfnum,, countread );
      IF <> THEN CALL ABEND;

        ! open the primary's files.
      CALL OPEN ( fnamel, fnuml, flagsl, sync^depthl );
      IF < THEN .... ! error.
      CALL OPEN(fname2,fnum2,flags2,sync^depth2)
      IF < THEN .... ! error.
        .
        .
      CALL OPEN ( fnamen, fnumn, flagsn, sync^depthn );
      IF < THEN .... ! error.

        ! monitor the backup cpu.
      CALL MONITORCPUS ( %100000 '>>' backup^cpu );

        ! create the backup process.
      CALL createbackup ( backup^cpu );
    END
  ELSE ! i'm the backup
    BEGIN
        ! save the primary's cpu num.
      backup^cpu := ppdentry[3].<0:7>;
        ! monitor the primary cpu.
      CALL MONITORCPUS ( %100000 '>>' backup^cpu );
      CALL CHECKMONITOR;
      CALL ABEND;
    END;

  ! read $RECEIVE.
  CALL READ ( rfnum, rbuf, count );

  ! execute the main program loop.
  CALL main^loop;
```

```
END;
```

The following is the example code in the "createbackup" procedure:

```
PROC createbackup (backup^cpu);
   INT backup^cpu;

  BEGIN
    .
    INT .pfile[0:11],
        pname[0:3],
        backup^pid[0:3],
        error;

    .
    CALL PROGRAMFILENAME ( pfile );

    CALL GETCRTPID ( MYPID, pname );

    CALL NEWPROCESS ( pfile,,,backup^cpu,backup^pid,error,pname );

    IF backup^pid THEN ! it was created.
      BEGIN
        backup^up := 1;
        CALL CHECKOPEN(receive,rfnum,1,,,,error) ! $RECEIVE file.
        IF <> THEN ... ! error.
        CALL CHECKOPEN( fname1,fnum1,flags1,sync^depth1,,,error );
        IF <> THEN ... ! error.
        CALL CHECKOPEN( fname2,fnum2,flags2,sync^depth2,,,error );
        IF <> THEN ... ! error.
          .
          .
        CALL CHECKOPEN( fnamen,fnumn,flagsn,sync^depthn,,,error );
        IF <> THEN ... ! error.

        CALL CHECKPOINT (,, fnum1,, fnum2,, .......,, fnumn );
        CALL CHECKPOINT (, addr, count, ... );
          .
          .
        IF (status := CHECKPOINT(stackbase)) THEN ! restart point.
           CALL analyze^checkpoint^status ( status );
      END; ! open files
  END; ! of createbackup
```

Process Startup for Non-Named Process Pairs

The startup for non-named process pairs is nearly identical to that for named process pairs, except for the following items:

- The determination of primary/backup designation.
- The primary must send a startup message to the backup.

- The backup must call the STEPMOM procedure for the primary. This is necessary because the checkpointing facility uses the creator process ID in the primary's Process Control Block to determine the destination of checkpoint messages.

- The startup message must be read via the READUPDATE procedure (and, therefore, replied to via the REPLY procedure). This is done so that the primary process will be suspended (and therefore prevented from checkpointing) until the backup calls the STEPMOM procedure.

Note: There is no "ancestor" relationship between a non-named process pair and the process initially responsible for their creation.

In the following list of the general steps involved in process startup, the differences from the startup process for named process pairs are indicated by lettered steps:

1. Save the stack base address for checkpointing.

2. Call ARMTRAP, so process will abend if trap occurs.

A. Open $RECEIVE (no-wait, receive depth = 1) and read the startup message via READUPDATE.

B. Determine if the process is the primary or backup -

```
     If primary then
       begin
C.       reply to startup message
5.       open files
6.       monitor the backup cpu
7.       create backup process:
           if created then
             begin
D.             send non-standard startup message to backup
8.             open files in backup process
9.             checkpoint environment to backup
             end
       end
E.   else ! backup ! monitor the primary.
```

12. Initiate a read on $RECEIVE to check for backup stopped or processor up/down messages.

After performing these steps, execute the main program loop.

A. READ STARTUP MESSAGE: The $RECEIVE file should be opened with no-wait i/o and <receive depth> >= 1 specified. <receive depth> >= 1 is specified so that the startup message can be read via a call to READUPDATE, then later replied to via a call to REPLY. This is necessary so that the backup process, after it reads its startup message, can cause the primary process to be suspended until it has a chance to call the STEPMOM procedure on the primary process.

```
INT .receive[0:11] := ["$RECEIVE", 8 * ["  "]], ! global
     rfnum,                                     ! variables.

 .
CALL OPEN ( receive, rfnum, 1, 1 );
IF < THEN CALL ABEND;
```

Next, the startup message (e.g., Command Interpreter parameter message) is read:

```
CALL READUPDATE ( rfnum, buf, count );
IF <> THEN CALL ABEND;
CALL AWAITIO ( rfnum,, countread );
IF <> THEN CALL ABEND;
```

The call to READUPDATE causes the sender of the startup message to be suspended until the message is replied to. At this point, a check should be made to determine if the message is a valid startup message (i.e., first word of the message = -1).

B. DETERMINE IF PRIMARY OR BACKUP: A recommended way to designate whether a non-named process is a primary or its backup, is to have the primary process send a non-standard startup message to the backup after the backup's creation. Then, if the new process reads a standard startup message, it knows that it is the primary; otherwise, it knows that it is the backup. A recommended form for a non-standard startup message is

```
<startup message>[0] = -1
<startup message>[1] = -2
```

The first word of the startup is the same as the Command Interpreter's startup message (this allows the program logic for checking for a valid startup message to be the same for both the primary and the backup). The primary/backup designation is made by checking word[1] of the startup message:

```
IF buf[ 1 ] <> -2 THEN ! i'm the primary
  BEGIN
    .
```

A startup message from the Command Interpreter contains the "default volume/subvol" names starting in word[1]. Therefore, word[1].<0:7> = "$" for a standard Command Interpreter startup message.

C. REPLY TO STARTUP MESSAGE: The primary process must reply to the startup message via a call to the REPLY procedure:

```
CALL REPLY;
```

permits the Command Interpreter to continue executing.

D. SEND NON-STANDARD STARTUP MESSAGE TO BACKUP: A non-standard startup message is sent to the backup following the backup's creation. The non-standard startup message provides the primary/secondary designation for the process pair:

```
IF backup^pid THEN ! it was created.
  BEGIN

    ! open a file to the backup process.
    CALL OPEN ( backup^pid, fnum );
    IF <> THEN ... ! couldn't open backup.  Bad news.

    ! build non-standard startup message.
    buf [ 0 ] := -1;
    buf [ 1 ] := -2;

    ! send the startup message.
    CALL WRITE ( fnum, buf, 4 );
    IF <> THEN ... ! couldn't write to backup.  Bad news.

       The primary process is suspended at this point until
       the backup process replies to the startup message.

    ! close the file to the backup process
    CALL CLOSE ( fnum );

    backup^up := 1;

    ! open files for backup process.
    CALL CHECKOPEN( fnamel,fnuml,flagsl,sync^depthl,,,error );
    IF <> THEN ... ! error.
      .
```

E. MONITOR THE PRIMARY: This is the action taken by the process if it is the backup. First, STEPMOM is called for the primary process (this is necessary so that the backup process will receive the checkpoint messages sent when the primary calls CHECKPOINT). Next, REPLY is called to reply to the startup message (this allows the primary to resume execution and make its first call to CHECKPOINT). Then, MONITORCPUS is called for the primary's processor module (this is done so that the primary's processor module will continue to be monitored if and when the backup takes over). The actual monitoring of the primary is accomplished by calling the CHECKMONITOR procedure:

```
ELSE ! i'm the backup
   BEGIN
     CALL MOM ( backup^pid );
     CALL STEPMOM ( backup^pid );
     IF < THEN CALL ABEND;
     CALL REPLY;
       ! save the primary's cpu num.
     backup^cpu := backup^pid[3].<0:7>;
       ! monitor the primary cpu.
     CALL MONITORCPUS ( %100000 '>>' backup^cpu );
     CALL CHECKMONITOR;
     CALL ABEND;
   END;
```

The backup process only returns from the call to CHECKMONITOR if the primary has not checkpointed its data stack. The primary checkpoints its stack for the first time at the end of creation of the backup process.

Main Processing Loop

In addition to normal transaction processing, the main processing loop must

1. checkpoint at appropriate points
2. check the $RECEIVE file for system messages
3. perform special action when taking over from the primary

FILE OPEN

Files are opened in a primary process via calls to the

OPEN procedure.

For disc files, when automatic path error recovery is desired, the number of write operations whose outcome the system is to remember is specified in the <sync depth> parameter to OPEN.

Files are opened in a backup process by its primary process via calls to the

CHECKOPEN procedure.

The use of CHECKOPEN permits both members of a process pair to have a file open, while retaining the ability to exclude other processes from accessing a file. For disc files open in this manner, a record/file lock by the primary is also an equivalent lock by the backup.

Note that the same parameter values that are passed to OPEN are also passed to CHECKOPEN; both files must be open with the same <file

number>, <flags> value, and <sync depth> value.

For example, in the primary process:

```
LITERAL
     flagsl       = ...,
     sync^depthl = ...;

INT .fnamel[0:11],
     fnuml,
     error;

     .
  ! open the file for the primary.
CALL OPEN ( fnamel, fnuml, flagsl, sync^depthl );
IF <> THEN ... ! error occurred.
     .
  ! open the file for the backup.
CALL CHECKOPEN ( fnamel, fnuml, flagsl, sync^depthl,,, error );
IF <> THEN ... ! error occurred.
```

CHECKPOINTING

Checkpoints are used to preserve transaction data and identify a restart point in the event of a failure. For each checkpoint in a primary process, there is a corresponding restart point in its backup process, as shown in figure 5-2.

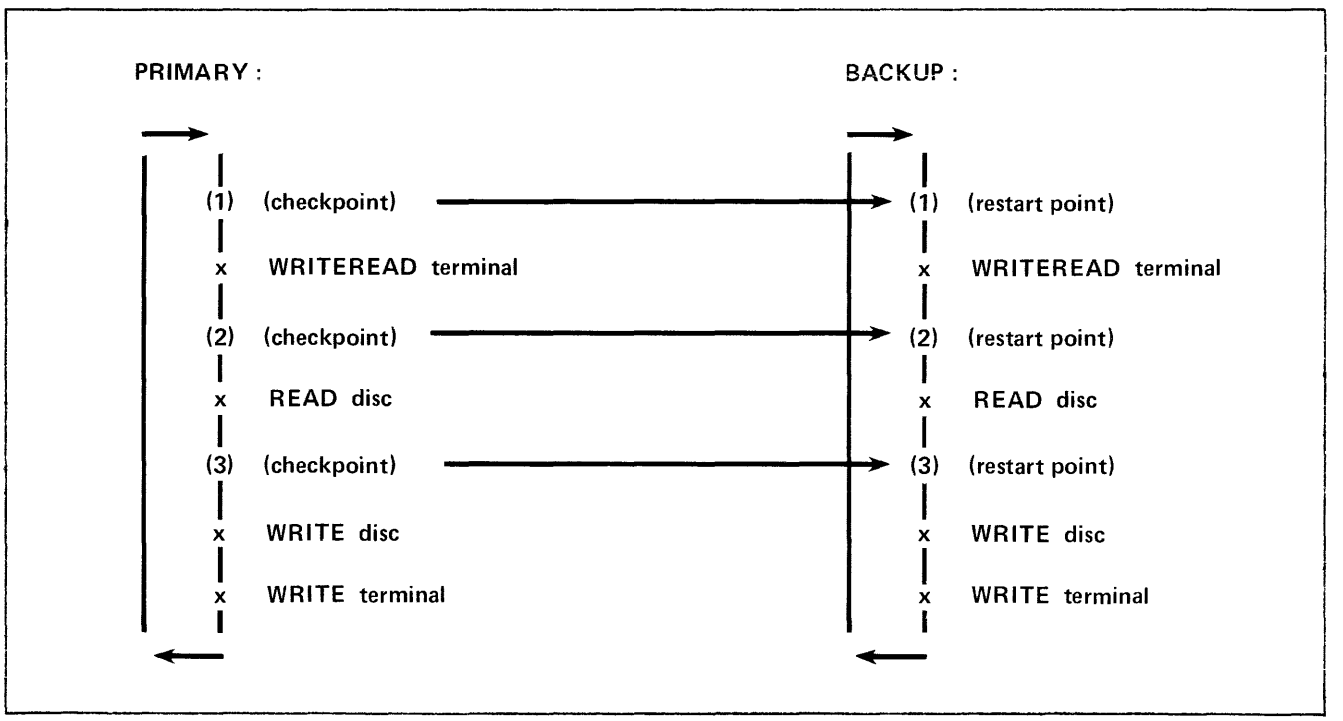


Figure 5-2. Checkpoints and Restart Points

For example, if a primary process fails subsequent to the return from a call to CHECKPOINT, its backup process will begin executing from the corresponding call to CHECKPOINT:

```
PRIMARY                                  BACKUP

   .                                     CALL CHECKMONITOR;
   .
   .
CALL CHECKPOINT(stk,,fnuma);             CALL CHECKPOINT(stk,,fnuma);

CALL WRITE(fnuma,...);                   CALL WRITE(fnuma,...);


   a failure of the primary past
   this point causes the backup to
   begin processing at this point
```

Enough checkpoints must be provided, and each must contain enough information, so that in the event of the primary's failure, the backup can take over the process pair's duties while maintaining the integrity of any data involved in the current transaction.

The amount of checkpointing that must be performed depends on the degree of recoverability desired. As an extreme example, a primary process could, after execution of each program statement, send its entire data area and its current program counter setting. A program of this type is recoverable after each statement. However, the amount of system resources needed to perform this type of checkpointing would be tremendous (a checkpoint following each statement).

In practice, however, checkpointing of internal calculations is not necessary, as they can be performed with virtually no loss of system throughput. In general, checkpointing is necessary only when data is being transferred between the internal program environment and a file. For example, the primary process may checkpoint the data just read from a terminal so that, if a subsequent failure occurs, the terminal operator won't have to reenter the data.

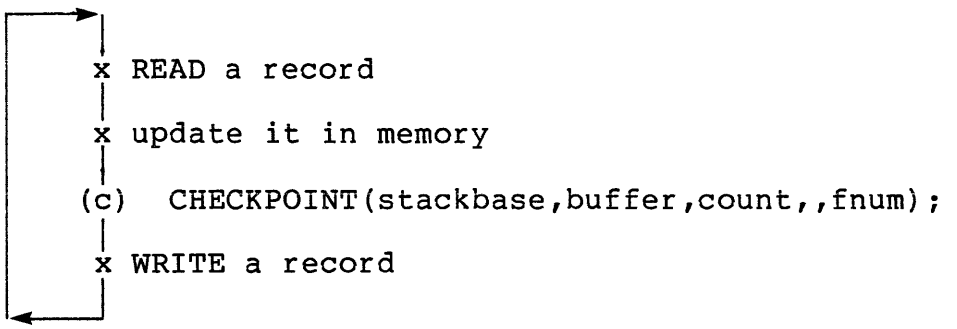
Guidelines For Checkpointing

As a general rule, a call to CHECKPOINT should immediately precede:

- Any write to a file (including a WRITEREAD to a terminal)
- A call to CONTROL or SETMODE for a file

To provide a greater degree of recoverability, a call to CHECKPOINT may immediately follow:

- A read from a terminal.



The call to CHECKPOINT should checkpoint the following information:

- A value or set of values indicating the program state. This is usually accomplished by checkpointing the process's data stack.
- If the checkpoint precedes a write to disc file, the file's Sync Block.
- The file's data buffer. Note that if the data buffer is within the memory stack area (i.e., from the application-defined stack base through the address indicated by the current S register setting), the data buffer will be checkpointed when the stack is checkpointed.

Adherence to the above guidelines assures that an application program can recover from disc file operations and, in most cases, terminal operations.

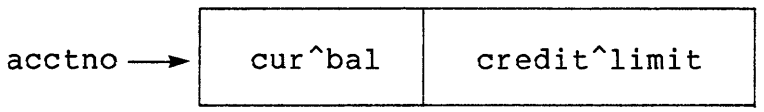
Also, as a general rule, the programmer should strive to keep the number of checkpoints in a processing loop and the amount of data checkpointed in a given call to CHECKPOINT to a minimum. An approach is to checkpoint only a portion of the program state (i.e., some data buffers and/or data stack) at one time. In this case, however, the programmer must be careful that any checkpoint which is also a restart point (i.e, includes the data stack) yields a valid program state. The programmer must also be careful, when checkpointing a data buffer without checkpointing the data stack, that the preceding restart point is still valid (i.e., does not use the new value of the data buffer). From the above, it should become apparent that proper checkpointing can be achieved only by careful analysis of the operation being performed and of the intended checkpoints and their contents.

Note also that i/o to non-disc and non-terminal devices involves very application-dependent recovery procedures. For example, a report to a line printer may have to be restarted from the last page, or a magnetic tape may have to be repositioned.

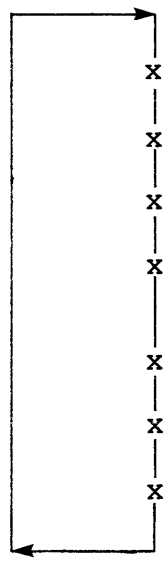
Example of Where Checkpoints Should Occur

The following is an example of a simple transaction cycle to update a record.

The record:

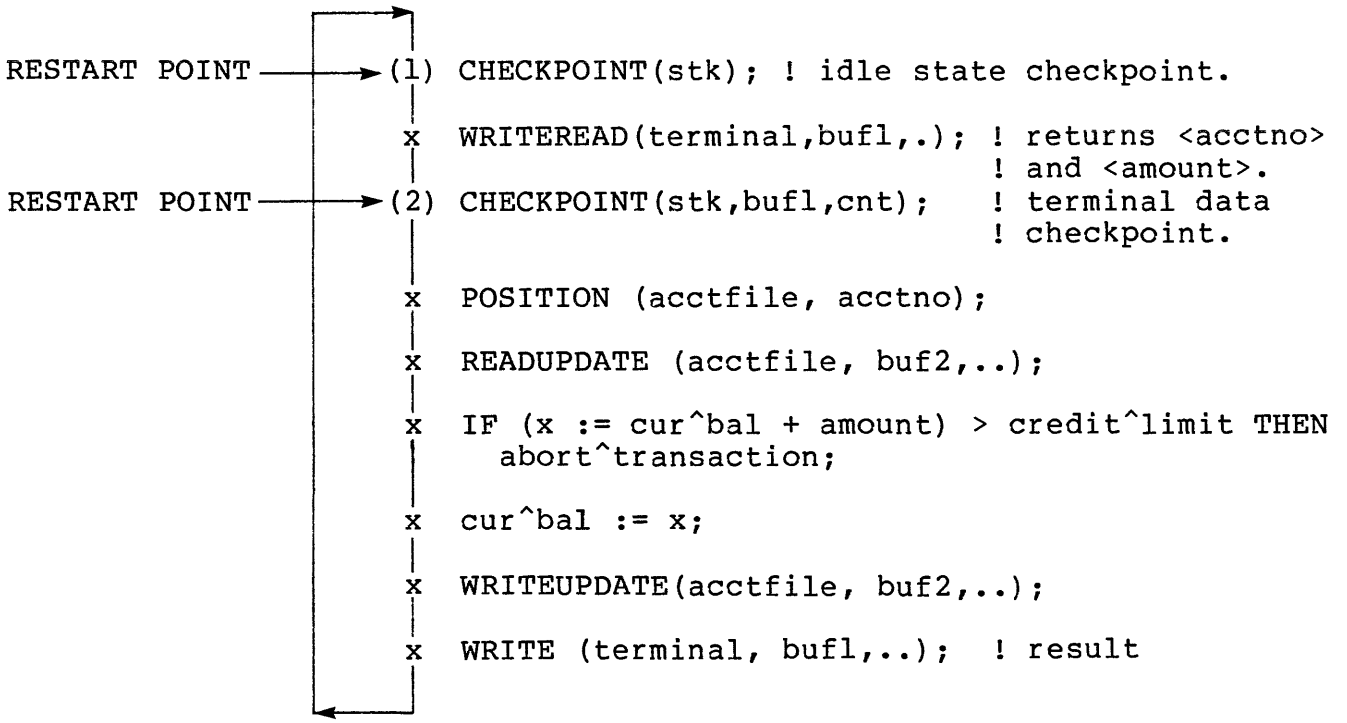


The transaction cycle (without checkpoints):



```
x  WRITEREAD(terminal,bufl,.); ! returns <acctno>
                               ! and <amount>.
x  POSITION (acctfile, acctno);

x  READUPDATE (acctfile,buf2,..);

x  IF (x := cur^bal + amount) > credit^limit THEN
     abort^transaction;

x  cur^bal := x;

x  WRITEUPDATE(acctfile, buf2,..);

x  WRITE (terminal, bufl,..);  ! result
```

The transaction cycle with insufficient checkpoints:



In this example, the first checkpoint identifies the program state as being idle (or waiting from input from the terminal). The actual checkpoint message consists of only the primary process's data stack.

The second checkpoint identifies the program state as "terminal entry just read". The checkpoint message consists of two parts:

1. the primary's data stack
2. the data read from the terminal

Here the assumption is that, because the transaction is driven by the data read from the terminal, this data is ample for the backup to perform the identical operation. This assumption is incorrect, however. A problem occurs if a failure occurs just following the WRITEUPDATE of the "acctfile". This is illustrated in the following transaction:

```
WRITEREAD(terminal, bufl,..); returns: "acctno" = "12345",
                                       "amount" = "$10"

(2) checkpoint "12345, $10"

POSITION (acctfile, 12345D);

READ (acctfile,buf2,..);
```

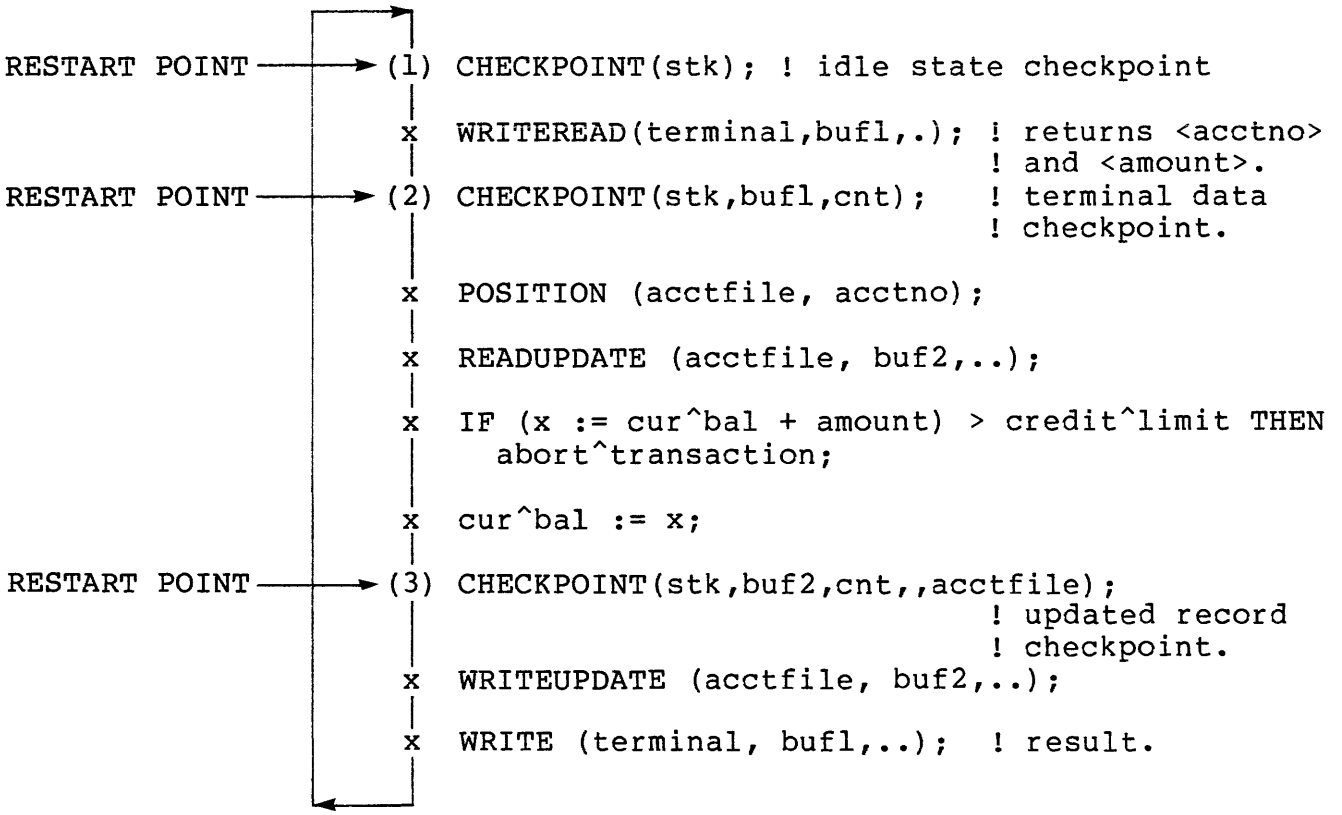
```
                    returns: "acctno"    "cur^bal"    "credit^limit"
                                 12345 ---->  $485  |  $500

IF (x := $485 + $10) > $500 THEN ...

cur^bal := x;

WRITEUPDATE (acctfile,buf2,..);

                     writes: "acctno"    "cur^bal"    "credit^limit"
                                 12345 ---->  $495  |  $500

****** FAILURE HERE ******
```

Backup's restart with latest checkpoint data: "12345, $10"

```
POSITION (acctfile, 12345D);

READ (acctfile,buf2,..);

                   returns: "acctno"    "cur^bal"    "credit^limit"
                                 12345 ---->  $495  |  $500

IF (x := $495 + $10) > $500 THEN ...
```

Here the test fails because the update to the disc completed successfully and the "cur^bal" has already been updated. The terminal operator is given an indication that "acctno" 12345 has attempted to exceed its credit limit; therefore the purchase is refused. However, account 12345's balance reflects that a purchase was made.

The transaction cycle with sufficient checkpointing:



The additional third checkpoint, (3), identifies the program state as "preparing to write an updated disc record to the disc". The checkpoint consists of three parts:

1. the primary process's stack
2. the disc file's sync information
3. the updated record

If the primary process fails between checkpoints 1 and 2, the backup process reissues the WRITEREAD to the terminal. If the primary process fails between checkpoints 2 and 3, the backup uses the terminal entry and continues the processing of the transaction. If the primary process fails subsequent to checkpoint 3, the backup uses the latest checkpointing information to reexecute the write to disc.

Note that checkpoint (2) and its associated restart point could be omitted. If this were done, a failure between checkpoints (2) and (3) would require the operator to reenter the transaction.

Checkpointing Multiple Disc Updates

When performing a series of updates to one or more disc files, the checkpoint for those updates can be performed at one point in the program. The result is less system usage than that required for several checkpoints.

The program should be structured so that the series of writes needed to update a file are performed in a group. For each file to be checkpointed in this manner, the <sync depth> parameter value of OPEN is specified as the maximum number of calls to WRITE for the file that are made between checkpoints for the file. Then, when a file is about to updated by performing <sync depth> writes to the file, the file's "sync block" and the data buffers about to be written to the file are checkpointed. In any case, care must be taken to ensure the integrity of any data referenced.

Considerations for No-Wait I/O

When taking over from a failure of the primary, any no-wait operations initiated, but not completed by the primary before its failure, must be reinitiated by the backup.

For example:

```
CALL READ ( rfnum, rbuffer, count ); ! no-wait.
   .
   .
CALL WRITE ( fnuml, buffer, count );          ! no-wait.

CALL CHECKPOINT ( stackbase );

fnum := -1;
CALL AWAITIO ( fnum, .. );                ! wait on any completion.
```

If a failure occurs, the backup begins executing following the call to CHECKPOINT. However, there will be no outstanding i/o operations.

A solution may be to checkpoint before the i/o operations are initiated. However, in the case of $RECEIVE, because the process may wish to have a read continually outstanding, this may not be possible. For $RECEIVE, the read can be reinitiated when the backup takes over.

Action for CHECKPOINT Failure

If an "unable to communicate with backup" error occurs when checkpointing (CHECKPOINT.<0:7> = 1 on return), the primary process should stop the backup process. The primary process should then create a new backup process when the STOP system message (i.e., # -5) is received (see "System Message, Recommended Action"). If the checkpoint failure persists, the failure should be noted accordingly,

and the primary should stop the creation attempts. (See the "analyze^checkpoint^status" example procedure under the heading "Takeover by Backup" in this section.)

Note: A checkpoint failure of this type normally indicates a system resource problem. Either the application process checkpoints are unduly large, or the SHORTPOOL size in the processor module where the failure occurs is too small.

SYSTEM MESSAGES

The following system messages are related to recovery from process and processor module failures. Their formats, in word elements, are

- CPU Down Message. There are two forms of the CPU Down message:

```
<sysmsg>            = - 2
<sysmsg>[1]         = <cpu>
```

  This form is received if a failure occurs with a processor module being monitored. Monitoring for specific processor modules is requested by a call to the process control MONITORCPUS procedure.

  and

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

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

  Note: Following a takeover by a backup process because of a processor module failure, the backup process, if it is an ancestor process, can expect to receive the second form of the CPU Down message. This message is received when a descendant process [pair] of the backup no longer exists because of the failure. Note that one of these messages will be received for each descendant process [pair] of the backup that disappears because of the processor module failure.

- CPU Up Message

```
<sysmsg>            = - 3
<sysmsg>[1]         = <cpu>
```

  This message is received if a reload occurs with a processor module being monitored.

- Process Normal Deletion (STOP) Message

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

  There are two forms of the STOP message:

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

  This form is received by a deleted process's creator if the deleted process was not named, or by one member of a process pair when the other member is deleted.

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

  This form is received by a process pair's ancestor when the process name is deleted from the PPD. This indicates that neither member of the process pair exists.

- Process Abnormal Deletion (ABEND) Message

  This message is received if the deletion is due to a call to the process control ABEND procedure, or because the deleted process encountered a trap condition and was aborted by the operating system.

  There are two forms of the ABEND message:

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

  This form is received by a deleted process's creator if the deleted process was not named, or by one member of a process pair when the other member is deleted.

  ```
  <sysmsg>              = -6
  <sysmsg>[1] FOR 3     = $<process name> of deleted process [pair]
  <sysmsg>[4]           = -1
  ```

  This form is received by a process pair's ancestor when the process name is deleted from the PPD. This indicates that neither member of the process pair exists.

Recommended Action

The following is the recommended action when the above messages are received:

msg #	action by primary
-2, cpu down:	Ignore it. An exception to this is if the second form of the cpu down message is received; the "ancestor" process may desire to recreate the failed process [pair].
-3, cpu up:	Create the backup, etc.
-5, backup stopped:	This shouldn't happen, but if it does, create the backup.
-6, backup abended:	Create the backup, etc.
other message #:	Take application-dependent action.

Note: For system messages #5 and #6, the program should assure that the primary process does not loop continuously because of continually failing backup process.

Following a read of a system message, a read on the $RECEIVE file should be initiated.

The following is an example procedure which analyzes system messages and takes appropriate action:

```
PROC analyze^system^message;

  BEGIN
    CASE $ABS ( rbuf ) OF
      BEGIN
        ;     ! 0.
        ;     ! 1.
        BEGIN ! 2 = cpu down.
          backup^up := 0;
        END;
        BEGIN ! 3  = cpu up.
          stop^count := 0; ! this must be checkpointed.
          CALL create^backup ( backup^cpu );
        END;  ! 3.
        ;     ! 4.
        BEGIN ! 5 = backup stopped.
          backup^up := 0;
          stop^count := stop^count + 1;
          CALL create^backup ( backup^cpu );
        END;  ! 5.
        BEGIN ! 6 = backup abended.
          backup^up := 0;
          stop^count := stop^count + 1;
          CALL create^backup ( backup^cpu );
        END;  ! 6.
        OTHERWISE ! other system message.
          BEGIN
```

```
            .
            .
          END;
        END; ! case of system message.
      ! issue a read to $RECEIVE.
      CALL READ ( rfnum, rbuf, count );
    END; ! analyze^system^message.
```

Note the "stop^count" variable. "stop^count" is used to detect repeated backup process failures that are not due to processor module failures. Note also that the variable is cleared when a CPU Up message is received. Nonstop programs should include such a variable to ensure that the primary process does not loop, continually recreating its backup. If "stop^count" reaches a count of 10, then the problem should be noted (e.g., a console message should be logged), and no further attempt at creation should occur until the problem is corrected.

TAKEOVER BY BACKUP

The following is the recommended action by the backup when it takes over from the primary. The action taken is dependent on the reason for the takeover:

If return is from CHECKMONITOR, call ABEND (primary's stack has not been checkpointed).

If return is from CHECKPOINT, then:

reason (CHECKPOINT.<8:15>)	action
0, primary stopped:	Call STOP.
1, primary abended:	Create backup, open its files, etc.
2, primary cpu down:	None (this will be taken care of when a subsequent CPU Up system message is received).
3, primary called CHECKSWITCH:	None.
any except 0:	Issue a read on $RECEIVE.

The following example procedure analyzes the value returned from CHECKPOINT and takes appropriate action:

```
PROC analyze^checkpoint^status ( status );
   INT status; ! return value of CHECKPOINT.

  BEGIN
    INT .backup^pid[0:3];

    IF backup^up THEN ! analyze^it.
      CASE status.<0:7> OF
        BEGIN
          ;     ! 0 = good checkpoint.
          BEGIN ! 1 = checkpoint failure.
            ! find out if backup is still running.
            CALL MOM ( backup^pid );
            CALL GETCRTPID ( backup^pid[3], backup^pid );
            IF = THEN ! backup still running.
              BEGIN
                ! stop the backup.
                CALL STOP ( backup^pid );
                backup^up := 0;
              END;
          END;  ! 1.
          BEGIN ! 2 = takeover from primary.
            CASE status.<8:15> OF
              BEGIN
                ! 0 = primary stopped.
                CALL STOP;
                ! 1 = primary abended.
                BEGIN
                  backup^up := 0;
                  stop^count := stop^count + 1;
                  CALL create^backup ( backup^cpu );
                END;
                ! 2 = cpu down.
                backup^up := 0;
                ! 3 = primary called CHECKSWITCH.
                ;
              END; ! case of status.<8:15>.
            ! issue a read to $RECEIVE.
            CALL READ ( rfnum, rbuf, count );
          END;  ! 2.
          BEGIN ! 3 = bad parameter to CHECKPOINT
            CALL DEBUG;
          END;  ! 3.
        END; ! case of status.<0:7>.
  END; ! analyze^checkpoint^status.
```

See the "analyze^system^message" procedure in "System Message, Recommended Action", for an explanation of the "stop^count" variable.

OPENING A FILE DURING PROCESSING

The possibility exists, when files are opened after process startup, that a failure could occur during the file open. This could result in the backup process opening the same file twice. The following is a recommended procedure for opening a file during processing:

```
INT PROC fileopen (filename, fnum, flags, syncdepth);
   INT .filename, .fnum, flags, syncdepth;

  BEGIN
    INT error := 1;

    WHILE error DO
      BEGIN
        CALL OPEN ( filename, fnum, flags, syncdepth );
        IF <> THEN
          BEGIN
            CALL FILEINFO ( fnum, error );
            RETURN error;
          END;

           At this point, the file is open in the primary.

        IF ( status := CHECKPOINT ( stackbase , fnum , 1 ) ) THEN
           CALL analyze^checkpoint^error ( status );

        CALL FILEINFO ( fnum, error );

           If this is executed because of a takeover from the
           primary, error 16 ("file number has not been opened") is
           returned from the call to file info.  This will result
           in the "WHILE error" loop being reexecuted.

      END;

    IF backup^up THEN
      BEGIN ! open the file in the backup.
        CALL CHECKOPEN( filename,fnum,flags,syncdepth,,,error );
        IF < THEN
          BEGIN ! backup exists, but could not open the file.
            CALL CLOSE ( fnum );
            RETURN error;
          END;
      END;

   RETURN 0; ! successful open by primary and backup if it exists.
END; ! fileopen.
```

CREATION OF A DESCENDANT PROCESS (PAIR)

Like opening files during processing, the possibility exists during creation of a descendant process or process pair that a failure could occur. This could result in the backup process creating a process already created by the primary. The following is a recommended method for descendant process creation:

```
CALL CREATEPROCESSNAME ( pname );

   The system generates a unique process name.

IF ( status := CHECKPOINT(stackbase,pname,4) ) THEN
   CALL analyze^checkpoint^status ( status );
CALL NEWPROCESS ( progfile,,, cpu, desc^pid, error, pname );
IF error > 1 THEN
  IF error.<0:7> <> 8       ! process name error.
     AND error.<8:15> <> 10 ! can't communicate with sys mon! THEN
     BEGIN ! unable to create the process due to resource problem
      .    ! or coding error.
      .
     END;
  ELSE
```

The following is necessary only if the backup needs the actual <cpu,pin> of the descendant process:

```
      BEGIN  ! duplicate name error, caused by takeover by backup.
        ppdentry ':=' pname FOR 3;
        CALL LOOKUPPROCESSNAME(buf)
        IF < THEN ... ! process no longer exists.

        ! save descendant's process ID.
        desc^pid ':=' pname FOR 4;

        ! determine actual <cpu,pin> of descendant.
        IF ppdentry[3].<0:7> <> cpu THEN
          IF ppdentry[4].<0:7> = cpu AND ppdentry[4] <> 0 THEN
             desc^pid[3] := ppdentry[4]
          ELSE ... ! the process no longer exists in the cpu.
      END;
```

This section is intended for application programmers who do not wish to use the checkpointing facility, but want instead to write their own checkpointing routines.

The following topics are discussed in this section:

- Backup Open
- File Synchronization Information

BACKUP OPEN

"Backup open" is a form of file open that permits a file to be open concurrently by both the primary and backup of a process pair regardless of the exclusion mode set by the primary process (except that access and exclusion modes must be the same for both the primary and the backup process, and file security is still enforced). This is accomplished by passing two parameters to OPEN: the process ID of the primary process which already has the file open, and the file number that was returned to the primary when it opened the file. After this form of OPEN, the primary and backup share access to the file such that in the case of disc files, when one process locks the file, the file becomes locked on behalf of both. (See figure 5-3.)

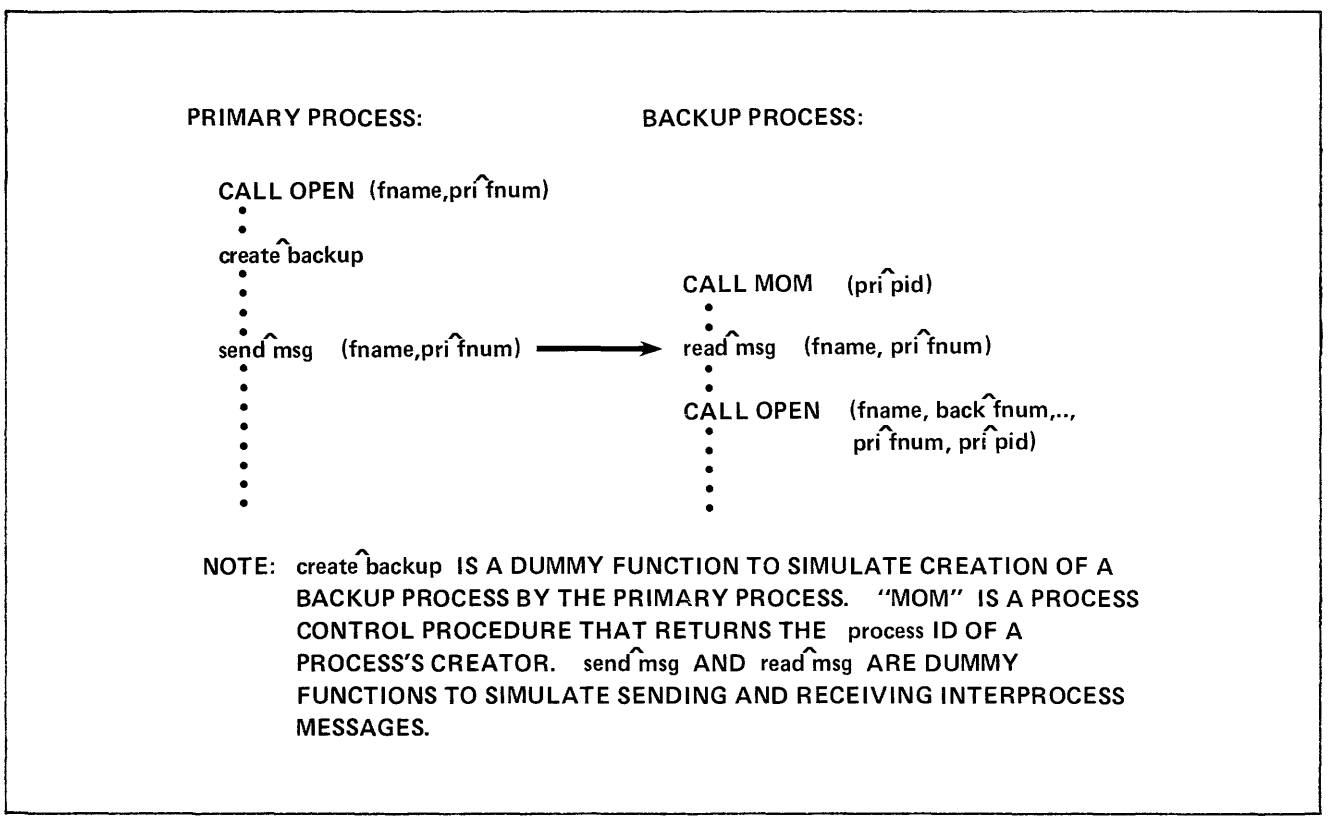


Figure 5-3. Backup Open by Backup Process

FILE SYNCHRONIZATION INFORMATION

File synchronization information is used by the system to determine if an operation by a backup process after a failure of its primary process is a new operation or a retry of an operation just performed by the primary.

The use of the sync information is accomplished in three parts:

1. Sync Depth

   The number of nonretryable operations that the file system is to "remember" is specified in the <sync depth> parameter to the OPEN procedure. This, normally, is the number of write operations that a primary process performs to a file between checkpoint messages to its backup.

   An example of opening a file and specifying a synchronization depth of one:

   ```
   CALL OPEN (fname, fnum, ,1);
   ```

   If opened by the backup process of a process pair, the primary file number and process ID must also be specified.

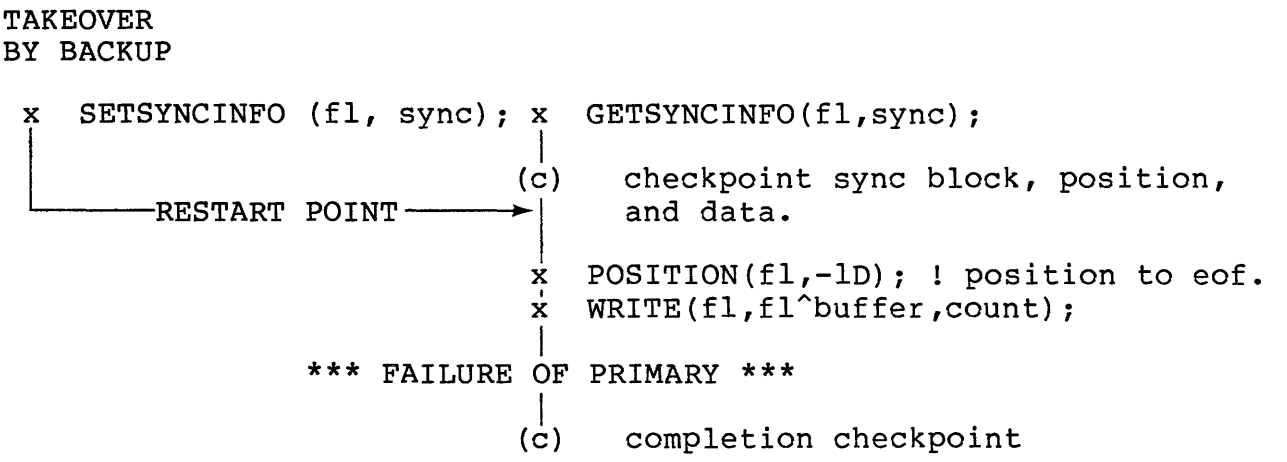
2. GETSYNCINFO Procedure

   When a primary application process is about to update a file by performing "synchronization depth" writes to the file, it first calls the GETSYNCINFO procedure, which returns "sync information" for the file. This information (which, incidentally, is never explicitly referenced by the application process) is then passed, along with the data to be written, in a checkpoint message to the backup application process. The primary process then performs the write operations, and upon completion informs its backup.

3. SETSYNCINFO Procedure

   If the primary application process fails, the backup process is notified by the operating system. Before attempting error recovery, the backup calls SETSYNCINFO with the sync information received in the latest checkpoint message. This synchronizes the retry operations that the backup is about to perform with any writes that the primary was able to complete before it failed. The backup then retries each write in the series (in the same order as the primary). If any operation was completed successfully by the primary, it is not performed by the file system; instead, just the completion status is returned to the backup process.

For example, in the following sequence of file system operations, a call to GETSYNCINFO precedes the file operations and a call to SETSYNCINFO precedes the restart point:



In this case, the write by the primary completed successfully, and the write by the backup when it takes over is ignored. The backup receives the completion status that the primary's failure.

Another procedure, RESETSYNC, is provided for cases where, after a failure, the backup process wishes to execute its error recovery by performing different operations than those of the primary, or where the backup process does not have a current synchronization block and the operations performed by the primary are not known. In either case, it is undesirable to have the file system mistakenly relate an operation performed by the backup to a different operation which was performed by the primary. By calling RESETSYNC after taking over for the failed primary process, the backup process ensures that this does not occur.

A call to RESETSYNC causes a RESETSYNC system message (system message -34) to be sent to the paired-access process file referenced in the call, indicating that the sync ID for that file has been reset to zero. Upon receipt of this message (receipt of RESETSYNC messages must be enabled by setting OPEN <flags>.<1> = 1 when opening the file), a server process using the sync ID mechanism should clear its local copy of the sync ID value.

SECTION 6

TRAPS AND TRAP HANDLING

TRAPS

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

trap no.	description
0	= illegal address reference
1	= instruction failure
2	= arithmetic overflow
3	= stack overflow
4	= process loop timer timeout
11 (%13)	= memory manager read error
12 (%14)	= no memory available
13 (%15)	= uncorrectable memory error
14 (%16)	= map parity error (NonStop systems only)

- Illegal Address Reference - an address was specified that was not within either the virtual code area or the virtual data area allocated to the the process. Virtual code area allocation is determined by the size of the program's code area. By default, virtual data area allocation is determined by the TAL compiler to be equal to the number of memory pages needed for the program's global storage plus one memory page for the program's data stack. The size of the virtual data area can be increased via the ?DATAPAGES command of the TAL compiler, the MEM parameter of the Command Interpreter RUN command, or the <memory pages> parameter of the NEWPROCESS procedure.

- Instruction Failure - an attempt was made to execute a code word that is not an instruction; an attempt was made by a non-privileged process to execute a privileged instruction; or on NonStop II systems, an illegal extended address reference was made.

- Arithmetic Overflow - the Environment Register "overflow" bit, ENV.<10>, is a "1" and the Environment Register "traps enabled" bit, ENV.<8>, is a "1". The overflow bit is set to "1" by the

hardware if the result of a signed arithmetic operation could not be represented with the number of bits available for the particular data type. Arithmetic overflow also occurs if a divide with a divisor of zero is attempted. Note that the overflow bit in the ENV register is not automatically cleared. If the application process is to recover from the overflow condition, it must specifically clear the ENV register overflow bit (otherwise, another overflow trap will occur).

The "traps enabled" bit of the ENV register is set to "1", by default, when a new process is created. If the process does not want a trap to occur when an overflow condition occurs, then the process must clear the traps enabled bit. This can be accomplished by executing the following TAL statements:

```
STACK 7; CODE ( SETE );
```

sets the Register Stack Pointer, ENV.<13:15>, to seven (7); clears ENV.<8:12>. ENV.<0:7> are not affected.

- Stack Overflow - an attempt was made to execute a procedure or subprocedure whose (sub)local data area extends into the upper 32K of the data area. Stack overflow also occurs, when calling an operating system procedure, if there is not enough remaining virtual data space for the procedure to execute (the procedure does not execute). The amount of virtual data space available is the lesser of 'G'[32,767] and the upper bound of the process's virtual data area (i.e., number of data pages specified when the process was created). Operating system procedures require approximately 350 words of user data stack space to execute.

- Process loop timer timeout - occurs only if the process has enabled "process loop timing" by making a call to the SETLOOPTIMER process control procedure. This trap indicates that the new time limit specified in the latest call to SETLOOPTIMER has expired.

- Memory Manager Disc Read Error - indicates that a hard (i.e., (unrecoverable) read error occurred while attempting to bring a page in from virtual memory.

- No Memory Available - indicates that a page fault occurred but no physical memory page is available for overlay.

- Uncorrectable Memory Error - indicates that an uncorrectable memory error was detected.

- Map Parity Error (NonStop systems only) - indicates that a parity error was detected by the memory map hardware when a memory reference was made.

TRAP HANDLING

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

If a trap condition is detected, one of three courses of action is taken:

1. If a process has previously made a call to the ARMTRAP procedure, control is transferred to the process's own trap handling mechanism.

2. If the process has not provided its own trap handler, the DEBUG procedure is called for the application process by the operating system.

3. If a trap has occurred and another trap occurs before the process can call ARMTRAP again, the process is deleted, and the creator of the process is sent a message indicating that an abnormal deletion occurred.

On NonStop systems, if DEBUG is entered because of a trap, the reason can be found in the current 'S'[-1:0] locations (use the DEBUG command "D S-1,2"):

'S'[-1] = trap number
'S'[0] = S register setting at the time of the trap

On NonStop II systems, DEBUG automatically displays the trap number; it is not available in the S register.

On both types of system, the P, ENV, and L register settings at the time of the trap can be determined by displaying the register contents (use the DEBUG command "D"). On NonStop II systems, the S register contents are also displayed. On NonStop systems, if the trap occurred while code in the system map was being executed, the process's stack is cut back until the first user stack marker is found. In this case, the value found in 'S'[0] is %177777.

Note: On NonStop systems, if DEBUG is entered because of a trap, DEBUG sets the S register to

$MIN (LASTADDR,32767) - 400

- that is, to 400 words below the last available location in the application process's data stack. The region from the S register setting to the last available location is used by DEBUG to call other operating system procedures. If the process's L register pointed into this region at the time of the trap, then referencing L-relative locations with DEBUG produces meaningless results.

The ARMTRAP procedure is used to specify an entry point into the application program where execution is to begin if a trap occurs. A number representing the type of trap, as well as a stack marker of the environment where the error occurred, is passed to the application process.

The call to the ARMTRAP procedure is:

```
CALL ARMTRAP ( <trap label> , <trap address> )
---- ------- - ------------ - -------------- -
```

where

<trap label>, INT:value,

is a label (non-zero P register value) to initially arm the trap mechanism. The label identifies a statement in the program where control is to be transferred if a trap occurs.

is zero (0) to rearm the trap mechanism after a trap has occurred and to cause the process to be re-launched. If this is specified, the process's registers at the time of the re-launch are set to the values indicated by the following 'L' relative locations:

```
'L'[-3] = new value for S register
'L'[-2] = new value for P register
'L'[-1] = new value for 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
```

<trap address>, INT:value,

is an address specifying the local data area for the application process's trap handler. This also indicates where the trap number and stack marker at the time of the trap are to be passed to the application process. After a trap occurs, 'S' and 'L' are set to <trap address> plus 5, and the five words starting at <trap address> plus 1 are (given relative to the new 'L' setting):

→

```
        '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
                                   14 = map parity error
                                        (NonStop systems only)

        'L'[-3] is the value of 'S' at the time of the trap; it
                is %177777 if the trap occurred while executing
                in the system code map
        'L'[-2] is the value of 'P' at the time of the trap
        'L'[-1] is the value of 'ENV' at the time of the trap
        'L'     is the value of 'L' at the time of the trap

      If <trap address> is passed as a value < 0, then any trap
      will result in the process being stopped with an abnormal
      deletion indication (i.e., ABEND message).

  example:

     CALL ARMTRAP ( @trap, @trap^addr );
```

CONSIDERATIONS

- If the trap handler is to call any operating system procedures, 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.

- The trap handler data area should not be located below the memory stack pointer, since the area below the stack pointer may be used internally by the operating system before ARMTRAP is called. Some programs which do so may operate correctly on NonStop systems but fail on NonStop II systems.

- 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 that 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 the next item).

- The Register Stack registers (i.e., R0-R7) upon entry to the application process's trap handler contain the values that they had at the time of the trap. To save these values, the first statement of the trap handler must be

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

which will save the Register Stack contents. Local storage may then be allocated by adding the appropriate value to 'S' via a statement of the form

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

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

- 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.

- The <trap label> must be in the same procedure as the call to ARMTRAP.

- If the application process's trap handler procedure is entered because of a trap, an exit from the procedure must be via a call to ARMTRAP with <trap label> specified as "0". The procedure must not exit through the stack marker at the current L register location (this would result in an invalid S register setting following the exit).

- 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 will occur immediately.

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

EXAMPLE

The following is an example of an application procedure that displays the current value of the P register when an arithmetic overflow trap occurs. Following an arithmetic overflow trap, the trap mechanism is re-armed, and the application process continues processing. If any other trap occurs, the procedure calls the DEBUG procedure.

The example trap handler procedure is:

```
PROC  overflowtrap;
  BEGIN
    INT  regs = 'L'+1,          ! R0-R7 saved here.
         wbuf = 'L'+9,          ! buffer for terminal i/o.
         preg = 'L'-2,          ! P register at time of trap.
         ereg = 'L'-1           ! ENV register at time of trap.
         trapnum = 'L'-4;       ! trap number.
    DEFINE  overflow = <10>#;   ! overflow bit in ENV register.
    STRING  sbuf = wbuf;        ! string overlay for i/o buffer.
    LITERAL  locals = 15;       ! # of words of local storage.

    ! arm the trap.
    CALL  ARMTRAP( @trap, $LMIN ( LASTADDR, %77777 ) - 400 );
    RETURN;

    ! enter here on a trap
trap:
    CODE (  PUSH %777;  ADDS locals  );

       saves R0-R7 and allocates local storage.

    IF  trapnum <> 2  THEN CALL DEBUG;

       calls DEBUG if the trap is not an overflow condition.

    sbuf ':=' "ARITHMETIC OVERFLOW AT %";
    CALL  NUMOUT( sbbuf[24], preg, 8, 6 );
    CALL  WRITE( home^term, wbuf, 30 );
    IF  <>  THEN ...

       formats and prints the message on the home terminal.
```

```
       ereg.overflow := 0; ! clear overflow.
       CALL  ARMTRAP ( 0, $LMIN ( LASTADDR, %77777 ) -400 );

          the overflow bit must be cleared before the old values of
          the registers are restored.

     END;
```

At the beginning of the program, the procedure is called:

```
          .
       CALL overflowtrap;
          .
```

From this point on, any arithmetic overflows are logged on the home terminal. For example, the following statement would cause the trap handler to be entered:

```
          .
       I := I/J;
```

if the current value of J was zero.

SECTION 7

SECURITY SYSTEM

This section discusses the following topics:

- General Characteristics of the Security System
- System Users
- Defining Users
- Logging on
- Passwords
- Accessor ID
- Disc File Security
- Licensing
- Interface to the Security System
- Network Security

GENERAL CHARACTERISTICS

The GUARDIAN security system is designed to fulfill four objectives:

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

However, the security system is designed so as not to interfere with application design in systems where security is not desired.

Additional security may be provided by the application program. Some examples of application program security checks are:

- Limitation of capability at a terminal

  It is not necessary to have a GUARDIAN Command Interpreter running at an application terminal. Therefore, the application program has control over what the terminal operator sees, and can limit the functions that he or she can perform.

- Physical security

  Programs that alter, or produce reports of, sensitive data may include routines that check the terminal from which they are run. This allows the application to restrict the running of the program to a specific terminal that is physically secure (for example, in a locked room for which there is only one key).

- Special devices

  These include authorization terminals such as badge readers, fingerprint readers, and so on.

SYSTEM USERS

Persons who have access to the system are called users. In general, there are four classes of users, with the following capabilities:

- Standard User

  A standard user is allowed to perform standard operations such as creating and purging disc files, running programs, displaying the system status, and so on. However, a standard user is limited as to the processes he or she can stop or debug.

- Group Manager

  A group manager user is allowed to add and delete users within the group, and to log on as any member of that group, as well as performing the standard user operations.

- System Operator

  A system operator user is allowed to reload processor modules, set the system time-of-day clock, and alter the operating state of the interprocessor buses, in addition to performing the standard user operations.

- Super ID

  The super ID user has total freedom to perform any operation in the system. This includes debugging privileged programs, accessing any file, logging on as any user without knowing the user's password, adding new user groups to the security system, running privileged programs that have not been licensed (see "Licensing"), and so on.

Additionally, for systems where security is not desired, all standard users can log on under the same standard user name. In a system like this, all users have equal access to all files in the system. Such a system must still have a super ID user, however, and perhaps a system operator user, so that their functions can be performed. Another alternative is for all users to access the system as the super ID.

DEFINING USERS

System users are defined to the system through the Command Interpreter ADDUSER command. For each user, a user name and a corresponding user ID must be specified:

```
ADDUSER <group name>.<user name> , <group id> , <user id>
        \_____________________/   \_____________________/
                user name                 user ID
```

The combination of <group name>.<user name> is referred to generically as a user name; similarly, the combination of <group id>,<user id> is referred to generically as a user ID.

Specifically, the form of a user name is:

<group name>.<user name>

- <group name> identifies an individual as a member of a group (a department, for example).
- <user name> identifies the individual within the group.

The form of a user ID, as a single numeric entity, is:

bits 0 through 7 = <group id> (in the range of 0 through 255)
bits 8 through 15 = <user id> (in the range of 0 through 255)

Assignment of user names and user ID's is entirely at the discretion of system management. Note, however, that a direct correspondence exists between <group name> and <group id>. This means that all users having the same <group id> must have the same <group name>. For example, a system may have the following groups defined:

<group name>	<group id>
ADMIN	1
MANUFACT	2
MARKETNG	3
.	.
.	.
.	.
SUPER	255

As many as 256 groups, with up to 256 users each, are possible.

The following group ID's and user ID's have special significance:

255 , 255 = Super ID (who is also a system operator and a group manager)
255 ,<255 = System operator
<255 , 255 = Group manager
0 , 0 = Null user

Only the super ID can define new groups. The super ID can define new users in any group. A group manager can define new users within his or her group only.

Execution of the ADDUSER command causes the new user's name and accessor ID to be entered into a file named $SYSTEM.SYSTEM.USERID. The Command Interpreter searches this file when the user logs on to relate the user name supplied in the LOGON command with a user ID, which is used internally in place of the user name. Note that only processes running as the super ID user are allowed access to this file (if it remains correctly secured).

As an example, assume the group ADMIN has previously been defined with a group ID of 1. To define a new user designated as ADMIN.BILL, with a user ID of 2 (which must be presently unassigned in group 1), the following ADDUSER command would be used:

```
ADDUSER ADMIN.BILL,1,2
```

This would cause the following entry to be made in the USERID file:

```
ADDUSER ADMIN.BILL,1,2
```

USERID FILE

USER NAME		USER ID
ADMIN	ANN	1,1
ADMIN	BILL	1,2
.	.	.
.	.	.
ADMIN	MANAGER	1,255

Naming Conventions

The following user names are conventionally given to the super ID user, the system operator user, and the null user.

User Class	User Name
super ID	SUPER.SUPER
system operator	SUPER.OPERATOR
null	NULL.NULL

LOGGING ON

Before a user can gain access to the system, he or she must log on. Logging on is accomplished by supplying the previously-defined user name to the system by means of the Command Interpreter LOGON command. For example, for a user defined as ADMIN.BILL to log on to the system, the following LOGON command would be given:

```
LOGON ADMIN.BILL
```

PASSWORDS

User names can be protected by passwords to prevent unauthorized individuals from accessing the system. A user defines his or her password by means of the Command Interpreter PASSWORD command. For example, the user ADMIN.FRED wants to specify a password. First, a LOGON command is executed to make ADMIN.FRED the current user:

```
LOGON ADMIN.FRED
```

Next, a PASSWORD command is executed and a password is specified:

```
PASSWORD KEEPOUT!
```

Whenever ADMIN.FRED logs on in the future, he must supply the password KEEPOUT! as part of the LOGON command.

```
LOGON ADMIN.FRED,KEEPOUT!
```

Executing a PASSWORD command without specifying a password removes the password protection from the current user.

ACCESSOR ID

Two accessor ID's are associated with a process: the creator accessor ID and the process accessor ID. The creator accessor ID identifies the user who initiated the creation of the process. The process accessor ID is normally the same as the creator accessor ID; however, it is the same as the process owner's user ID if file adoption has been specified for the related program file.

The security system uses the process accessor ID when determining if file access should be allowed (see "File Security"). In addition, the process accessor ID is used to determine whether certain restricted operations (STOP, DEBUG, and STEPMOM) can be performed by users other than a process's creator and the super ID. Users who are allowed to perform these operations are:

- The super ID
- A user process with a process accessor ID that is the same as that of the target process's creator
- A user process with a process accessor ID equal to the target process's accessor ID (this includes the caller to STEPMOM)

When a process is created, the operating system passes the process accessor ID to the descendant process. This ID becomes the creator accessor ID of the new process. The process accessor ID of the new process can come from either of two sources: from the process accessor ID of its creator (this is the usual case; see figure 7-1) or from the owner ID of the process's program file (for special file security applications; see "Adopting a Program File's Owner ID").

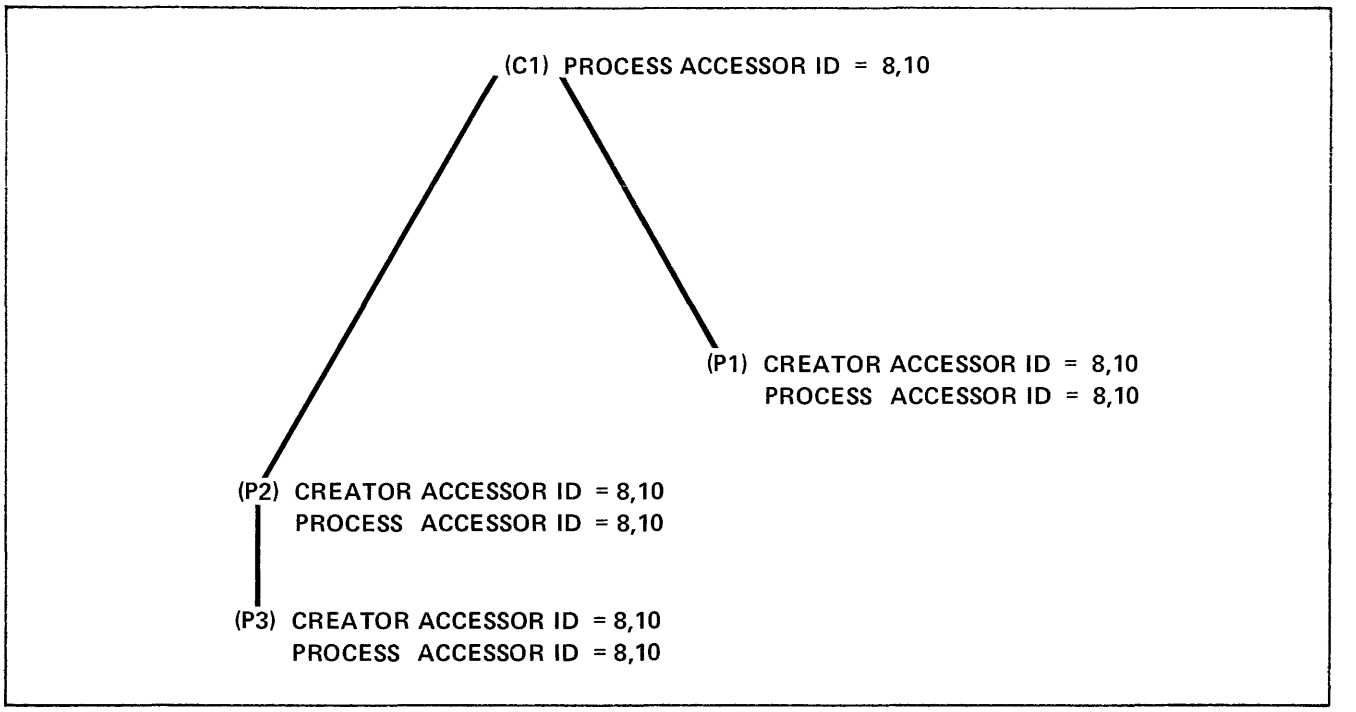


Figure 7-1. Passing of Accessor ID's

A process can obtain its creator accessor ID and process accessor ID via the CREATORACCESSID and PROCESSACCESSID procedures, respectively.

DISC FILE SECURITY

Each disc file has an owner, who is the user who created the file. A file's owner is identified by an owner ID, which is the same as the creating user's accessor ID.

Four types of access are allowed for a file: Read, Write, Execute (run), and Purge. For each type of access, the file's owner can specify the level of security that is to be enforced. Seven levels of security are available:

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

When a disc file is created, it is assigned the owner's current default security (controlled by the Command Interpreter DEFAULT and VOLUME commands). The security for a file may be changed via the file management SETMODE or SETMODENOWAIT procedure, or via the File Utility Program SECURE command. The following codes for security levels are used by the DEFAULT, VOLUME, and FUP SECURE commands:

"-" = local super ID only
"U" = member of owner's user class -- i.e., owner (local or remote)
"C" = member of owner's community -- i.e., member of owner's group (local or remote)
"N" = any user (local or remote)
"O" = owner (local)
"G" = member of owner's group (local)
"A" = any user (local)

When file opening is attempted, the local/remote attribute and the process accessor ID are used to determine the accessor's security level. If the opener is the local super ID, the security check is bypassed. Otherwise, the security level is determined in two steps:

1. The file's owner ID is compared with the opener's process accessor ID:

   a. If the opener is the file's owner, or the group manager (<group ID> is the same in both ID's and <user ID> of the process accessor ID = 255), the security level is 2.

   b. If the opener is not the owner, but is a member of the owner's group (<group ID> of both ID's are equal), the accessor's security level is 1.

   c. If the opener is any other user, the accessor's security level is 0.

2. If a remote process is making the access, 4 is added to the accessor's security level.

The security system then compares the accessor's security level with the file security level that has been specified for the requested access (read, write, execute, or purge). Table 7-1 shows the allowed accesses.

Table 7-1. Allowability of File Access

ACCESSOR'S SECURITY LEVEL	FILE SECURITY LEVEL 7	6	5	4	2	1	0
7	Y	Y	Y	Y	Y	Y	Y
6	-	Y	Y	Y	-	-	-
5	-	-	Y	Y	-	-	-
4	-	-	-	Y	-	-	-
2	-	Y	Y	Y	Y	Y	Y
1	-	-	Y	Y	-	Y	Y
0	-	-	-	Y	-	-	Y

For example, assume that a file owned by ADMIN.BILL has been secured by the FUP command

```
SECURE BILLFILE, "AGNU"
```

which specifies that any local user can read from the file, only local members of the ADMIN group can write to the file, any network user can execute the file, and only the owner can purge it. ADMIN.ANN, if she were operating via the network from a remote system, could do nothing more than execute the file, but if she were logged on locally she could also gain read or write access.

The Command Interpreter's DEFAULT command allows a user to specify the default file security for all files created by the user. This allows protection for new files to be applied automatically.

Adopting a Program File's Owner ID

This feature of the security system allows the owner of a program file (or the super ID) to specify that the process accessor ID of any process created from that program file is to be the same as the program file's owner ID instead of the creating process's process accessor ID. (See figure 7-2.) This adoption affects the files that the new process can access, and the "restricted" operations that can be performed on or by the process. Adoption is specified via the SETMODE and SETMODENOWAIT procedures and the File Utility Program SECURE command.

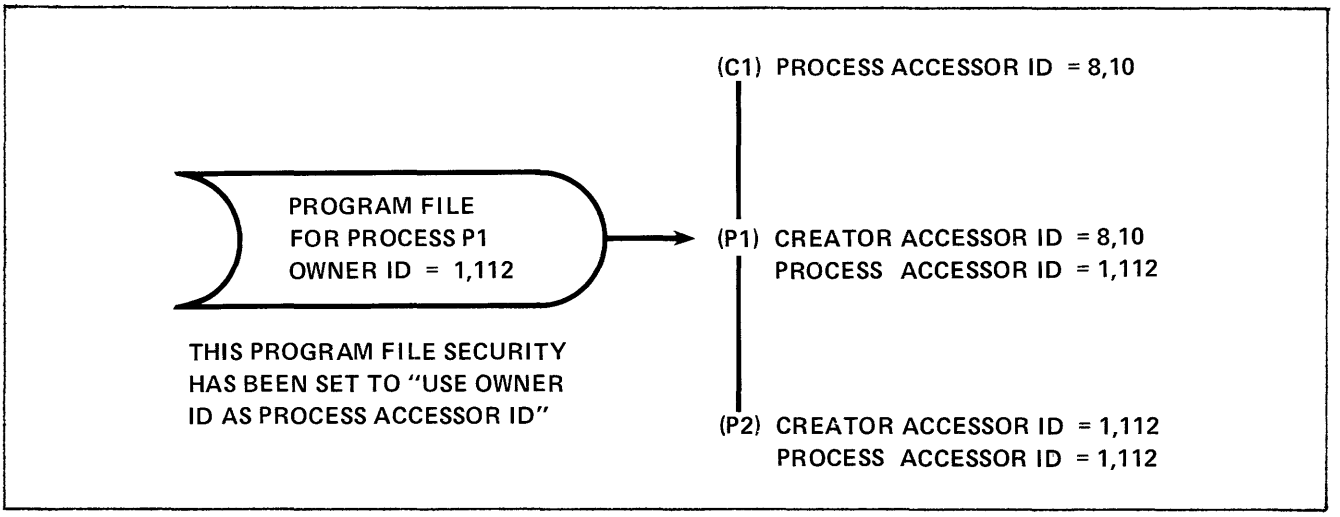


Figure 7-2. Effect of Adopting a Program File's Owner ID

This feature, along with the ability to change the ID of a file's owner, enables the application programmer to create files that are accessible only to certain programs.

For example, a record in an employee file may contain the employee's name, address, and salary, among other items of information. If normal file security were used, either the file or the program that accesses the file could be restricted to particular users. However, it may be desirable for all users to have file access for obtaining name and address data, but not the employee's salary. This could be done by having a program that accesses the file and returns the names and addresses only. The program file's owner ID and the data file's owner ID are the same; the program file's security permits any user to run it; the data file's security permits reading and writing by the owner only. When the query program is run, the new process's accessor ID is set to that of the program file owner's. Thus, although the program may be run by any user, it still provides controlled access to the data file.

LICENSING

If a program contains privileged procedures (procedures having the attributes CALLABLE or PRIV), it must be licensed before it can be run in the system (unless run by the super ID). Licensing can be performed only by the super ID via the File Utility Program LICENSE command.

Programs running in the privileged mode have total freedom to access operating system tables and to execute privileged instructions and procedures, so it is possible for such programs to circumvent the file security checks and thereby gain access to any file. However, some privileged programs are needed in the system (for example, the Command

Interpreter). Through licensing, the installation may run privileged programs that it has authorized, but users can not run unauthorized privileged programs.

NOTE: If a licensed file is opened with write access or read/write access, the file becomes unlicensed.

INTERFACE TO THE SECURITY SYSTEM

User interface to the security system can be established through any of three means:

- Through the Command Interpreter
- Through the File Utility Program (FUP)
- Through system procedure calls in user programs

Command Interpreter Interface

The following Command Interpreter commands provide the user interface to the Security System:

ADDUSER command (super ID and group manager only)
DEFAULT command
DELUSER command (super ID and group manager only)
LOGOFF command
LOGON command
PASSWORD command
REMOTEPASSWORD command
USERS command
VOLUME command

These commands are described in the "Command Interpreter" section of the GUARDIAN Operating System Command Language and Utilities Manual.

FUP Interface

The following FUP commands provide the user interface to the Security System:

GIVE command
LICENSE command (super ID only)
REVOKE command
SECURE command

These commands are described in the "File Utility Program" section of the GUARDIAN Operating System Command Language and Utilities Manual.

Programmatic Interface

The following procedures provide interface between user programs and the Security System:

- Security System Procedures:

  CREATORACCESSID
  PROCESSACCESSID
  USERIDTOUSERNAME
  USERNAMETOUSERID
  VERIFYUSER

- File System Procedures:

  SETMODE
  SETMODENOWAIT

- Process Control Procedure:

  SETSTOP

Security interfacing with these procedures is described in sec. 7.2.

Operational Limitations

The security system limits the operations a given user is eligible to perform, based on the classification of the user (see table 7-2).

Table 7-2. Operational Restrictions

Command Interpreter		Classification: Super ID User	Eligibility: Group Manager	Standard User
ADDUSER	Defines new user to security system.		x	x
	Defines new group/user to security system.			x
DEBUG	Debugs unprivileged process run by current user.	x	x	x
	Debugs any process, including privileged ones.			x
DEFAULT	Sets current user's default volume and subvolume names and default security.	x	x	x
DELUSER	Deletes user from security system.		x	x
	Deletes group from security system.			x
PASSWORD	Sets current user's LOGON password.	x	x	x
PAUSE	Allows <cpu,pin> form for designated process if both processes have same process accessor ID.	x	x	x
	Allows <cpu,pin> form in any case.			x
RUN	Starts a program.	x	x	x
	Starts an unlicensed privileged program.			x
STOP	Stops process run by current user (creator), or that has same process accessor ID as current user, or that has stop mode of 0.	x	x	x
	Stops any process.			x
USERS	Lists attributes of current user.	x	x	x
	Lists attributes of all users in current group.	x	x	x
	Lists attributes of all users in all groups.	x	x	x
VOLUME	Changes current user's current default volume and subvolume names and default security for this logon session only.	x	x	x

		Classification / Eligibility		
		Super ID User	Group Manager	Standard User
File Utility Program				
GIVE	Changes ownership of file owned by current user.	x	x	x
	Changes ownership of any file.			x
INFO	Lists a file's characteristics.	x	x	x
LICENSE	Permits privileged program to be run by users other than super ID.			x
REVOKE	Unlicenses privileged program.			x
SECURE	Sets file security of file owned by current user.	x	x	x
	Sets disc file security of any file.			x
File System Procedures				
CREATE	Creates a file with default security.	x	x	x
OPEN	Opens a file if security check is passed.	x	x	x
	Opens any file.			x
SETMODE and SETMODENOWAIT				
	Changes security of file owned by current user.	x	x	x
	Changes security of any file.			x
Security System Procedures				
CREATORACCESSID				
	Obtains process's creator accessor ID.	x	x	x
PROCESSACCESSID				
	Obtains process's process accessor ID.	x	x	x

Process Control Procedures		Classification: Super ID User	Group Manager	Standard User	Eligibility
STEPMOM	Changes creator process ID for designated process if both processes have same process accessor ID.	x	x	x	
	Changes creator process ID for designated process regardless of process accessor ID's.				x
STOP	Stops process started by creator, or that has same process accessor ID as creator, or that has stop mode of 0.	x	x	x	
	Stops any process.				x

NOTE: In the preceding descriptions, the "current user" is the process that invokes a command or procedure. The eligibility to perform operations is determined by checking the process accessor ID of that process.

NETWORK SECURITY

A user at system X wishing to access a file (disc file, device, or process) residing on system Y must satisfy each of the following three requirements:

- The user on system X must also be a user on system Y.
- The user on system X must know the correct remote password for accessing files on system Y.
- The user on system X must have the authority to access a disc file on system Y (explained previously under "File Security").

Global Knowledge of User ID's

Each user is known to the computer by a user name, such as ADMIN.BILL, and a user ID, such as 1,2. A user can access files on a system only if his user name and user ID are known to that specific system. So if ADMIN.BILL, whose user ID is 1,2, wishes to access a file on a remote system, that system must also have a user named ADMIN.BILL whose user ID is 1,2.

Remote Passwords

Once the user ID's of network users have been added to each node of the network, a system of remote passwords is used to specify whether remote access is permitted.

Each user ID has associated with it a set of remote passwords. One, specified with the command

```
REMOTEPASSWORD \<my system name> , <allow-access password>
```

designates the password required for a remote user to access this system. The others, specified by

```
REMOTEPASSWORD \<remote system name> , <request-access password>
```

define passwords used on attempts to access a remote system; the attempt is successful if the request-access password matches the allow-access password previously specified by the remote user.

Both types of passwords consist of up to 8 nonblank characters. Control characters are allowed, and lower-case characters are not upshifted.

Consider two systems, named \A and \B, in a network. At each system, a user named ADMIN.BILL, with user ID 1,2, has been defined.

At system \A, a user enters the commands:

```
LOGON ADMIN.BILL
REMOTEPASSWORD \A, SHAZAM
```

ADMIN.BILL's allow-access password is SHAZAM. In the future, any user who logs on at a remote system as ADMIN.BILL must specify SHAZAM as his request-access password to be able to access system \A. For example, at system \B, a user enters:

```
LOGON ADMIN.BILL
REMOTEPASSWORD \A, SHAZAM
```

This user now has remote access to system \A as ADMIN.BILL, and may now perform such operations as creating processes and accessing certain disc files.

A remote password, once defined, remains in effect until modified by a subsequent REMOTEPASSWORD command. ADMIN.BILL may log off and then log on again without having to respecify his remote passwords.

ADMIN.BILL, logged on at system \B, does not have quite the same status at \A as the ADMIN.BILL at \A. ADMIN.BILL at \B is a remote accessor of \A; consequently, he cannot access disc files on \A that are secured for local access only.

However, ADMIN.BILL at \A has no access at all to system \B. For ADMIN.BILL to gain access to \B, an allow-access password must be defined for ADMIN.BILL at \B, matched by a request-access password at \A. For example, at \B:

```
LOGON ADMIN.BILL
REMOTEPASSWORD \B, aardvark
```

and at \A:

```
LOGON ADMIN.BILL
REMOTEPASSWORD \B, aardvark
```

Now ADMIN.BILL at system \A can access system \B.

The following considerations apply to remote passwords:

- As shown previously, the absence of an allow-access password prevents remote access as that user. Thus, if MARKETNG.SUE does not supply an allow-access password, no remote user with the same user ID can access MARKETNG.SUE's system.

- The command

  REMOTEPASSWORD \<system name>

  removes any previously designated password (either for the local system or a remote one). The command

  REMOTEPASSWORD

  removes all remote passwords.

- A request-access password can be issued before the corresponding allow-access password. Remote access is permitted as soon as both remote passwords have been defined (provided they match).

- Remote passwords are independent of local passwords. In the preceding example, ADMIN.BILL could issue the command

  PASSWORD <local password>

  at either system to prevent unauthorized persons from logging on as ADMIN.BILL.

Process Access

Several security considerations relate to remote processes:

- With respect to a given system, each process in the network is either "local" or "remote." The following rules state that:

  - A process is remote if it is running in a remote system.

  - A process is remote if its creator is in a remote system.

  - A process is remote if its creator is remote.

  By the last two rules, even a process that is running in a given system may be remote with respect to that system. These rules prevent a remote process from creating another process to access a file whose security specifies local access only.

- A remote process cannot suspend or activate a local process. A remote process cannot stop a local process, unless the stop mode of the local process is 0 (anyone may stop it).

- A remote process cannot put a local process into DEBUG.

Programmatically Logging On

A process that is remote with respect to the system in which it is running can become local. For example, a user on system \A can become local in respect to system \B by starting a Command Interpreter in \B and logging on. The creator of the CI in \B is the user's CI in \A. According to the preceding rules, the CI in \B is remote with respect to \B, but the user's LOGON command causes that CI to become local with respect to \B. So the concept of local and remote users becomes equivalent to that of local and remote processes: A user is local (or remote) with respect to a given system if his Command Interpreter is local (or remote) with respect to that system. A process can make itself local with respect to the system in which it is running by programmatically logging on to that system. This is done by calling the VERIFYUSER procedure, which verifies a user's password and optionally allows a process to make the user's ID its own, thereby becoming local with respect to the system in which it is running.

The following procedures provide the interface between user programs and the security system:

- Security System Procedures

  CREATORACCESSID
  PROCESSACCESSID
  USERIDTOUSERNAME
  USERNAMETOUSERID
  VERIFYUSER

- File System Procedures

  SETMODE
  SETMODENOWAIT

- Process Control Procedure

  SETSTOP

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

The call to the CREATORACCESSID procedure is:

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

where

   <accessor id>, INT,

     is returned the accessor ID of the caller's creator.  It is
     returned in the form

       <accessor id>.<0:7>  = group ID
       <accessor id>.<8:15> = user ID

example:

   creatorid := CREATORACCESSID;
```

CONSIDERATIONS

- 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.

The PROCESSACCESSID procedure is used to obtain the accessor ID of the the calling process.

The call to the PROCESSACCESSID procedure is:

```
<accessor id> := PROCESSACCESSID
                 ---------------

where

   <accessor id>, INT,

     is returned the accessor ID of the caller.  It is returned
     in the form

       <accessor id>.<0:7>  = group ID
       <accessor id>.<8:15> = user ID

example:

   myaccessorid := PROCESSACCESSID;
```

CONSIDERATIONS

- For a given process, the accessor ID returned from the PROCESSACCESSID procedure is normally the same as that returned from the CREATORACCESSID procedure. The only time that the accessor ID's may differ is when a program file is run for which "set accessor ID to program file's owner ID" has been specified. In that case, the accessor ID returned by PROCESSACCESSID is the same as that of the program file's owner.

Functions for SETMODE and SETMODENOWAIT Procedures

The SETMODE and SETMODENOWAIT procedures can be used in a program to set and/or obtain a file's security and owner ID.

Note: Only a file's owner or the super ID is allowed to set file security or change the owner ID.

The SETMODE functions related to security are given in table 7-3.

Table 7-3. SETMODE Functions Related to Security

```
<function>

1 = set disc file security:

    <parameter 1>

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

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

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

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

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

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

        For each of the fields from .<4:6> through .<13.15>,
        the value may be any one of the following:

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

    <parameter 2> is not used.

2 = set disc file owner ID:

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

    <parameter 2> is not used.
```

Some examples:

First, to change a file's security setting:

```
LITERAL security = %0222;
  .
  .
CALL SETMODE ( fnum, 1, security );
IF < THEN ... ;
```

sets the file's security to

```
read = any
write  = owner
execute = owner
purge = owner
```

Second, to specify that the file's owner ID should be used as the process's accessor ID when the program file is run:

```
LITERAL prog^sec = %102202;
  .
  .
CALL SETMODE ( pfnum, 1, prog^sec );
IF < THEN ... ;
```

sets the file's security to

```
set accessor ID to owner's ID when file is run
read    = owner
write   = owner
execute = any
purge   = owner
```

Third, to change the file's owner ID:

```
INT owner^id;
  .
  .
owner^id.<0:7> := new^group^id;
owner^id.<8:15> := new^user^id;
CALL SETMODE ( fnum, 2, owner^id );
IF < THEN ... ;
```

sets the file's owner ID to the value specified in "owner^id".

Fourth, to obtain the file's current security setting:

```
INT filesec;
  .
  .
CALL SETMODE ( fnum, 1,,, filesec );
IF < THEN ... ;
```

returns the file's current security settings in "filesec".

Finally, to obtain the file's owner ID:

```
CALL SETMODE ( fnum, 2,,, owner^id );
IF < THEN ... ;
```

returns the file's owner ID in "owner^id".

The SETSTOP procedure permits a process to protect itself from being deleted by any process but itself or its creator.

The call to the SETSTOP procedure is:

```
{ <last stop mode> := } SETSTOP ( <stop mode> )
{ CALL               } ------- - ----------- -
```

where

<last stop mode>, INT,

is returned either the preceding value of <stop mode>, or -1 if an illegal mode was specified.

<stop mode>, INT:value,

specifies a new stop mode. The modes are:

0 = stoppable by any process

1 = stoppable only by

- the super ID
- a process whose process accessor ID = this process's creator
- a process whose process accessor ID = this process's accessor ID (this includes the caller to STEPMOM)

2 = unstoppable (privileged users only)

example:

```
last^mode := SETSTOP( new^mode );
```

CONSIDERATIONS

- The default stop mode when a process is created is 1.
- If a process's stop mode is 1 and a STOP is issued to it by a process without the authority to stop it, the process does not stop; it is deleted, however, if and when the stop mode is changed back to 0.

The USERIDTOUSERNAME procedure returns the user name, from the file $SYSTEM.SYSTEM.USERID, that is associated with a designated user ID.

The call to the USERIDTOUSERNAME procedure is:

```
CALL USERIDTOUSERNAME ( <id name> )
---- ---------------- - --------- -

where

   <id name>, INT:ref:8,

     on the call, 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}
       <id name>.<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>[3] 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 was returned.
   >  (CCG) indicates that the specified user ID is not defined.

example:

   id^name.<0:7>  := group^id;
   id^name.<8:15> := user^id;
   CALL USERIDTOUSERNAME ( id^name );
   IF <> THEN ... ;
```

The USERNAMETOUSERID procedure returns the user ID, from the file $SYSTEM.SYSTEM.USERID, that is associated with a designated user name.

The call to the USERNAMETOUSERID procedure is:

```
CALL USERNAMETOUSERID ( <name id> )
---- ---------------- - --------- -

where

   <name id>, INT:ref:1,

     on the call, 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>[3] 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.USERIDS
            file.
   =  (CCE) indicates that the designated user ID was returned.
   >  (CCG) indicates that the specified user name is not
            defined.

example:

   name^id    ':=' group^name FOR 4;
   name^id[3] ':=' user^name  FOR 4;
   CALL USERIDTOUSERNAME ( name^id );
   IF <> THEN ... ;
```

The VERIFYUSER procedure verifies, and optionally logs on, a user.

The call to the VERIFYUSER procedure is:

```
CALL VERIFYUSER ( <user name or id>
---- ---------- - -----------------
                  , <logon>, <default>, <default length> )
                                                         -
where

   <user name or id>, INT:ref:12,

      is an array containing either 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>, INT:value,

      if present, has the following meaning:

            0 : verify user, but do not log on
         <> 0 : verify user and log on

      if omitted, a value of 0 is understood.
                                                         →
```

```
   <default>, INT:ref:18,

      if present, is returned information regarding the user
      specified in <user name or id>:

         <default>[0:3]       = group name, blank filled
         <default>[4:7]       = user name, blank filled
         <default>[8].<0:7>   = group ID
         <default>[8].<8:15>  = user ID
         <default>[9:12]      = default volume, blank filled
         <default>[13:16]     = default subvolume, blank filled
         <default>[17]        = default file security, as follows:

            <default>[17].<4:6>   = read    } 0 = "A"  4 = "N"
            <default>[17].<7:9>   = write   } 1 = "G"  5 = "C"
            <default>[17].<10:12> = execute } 2 = "O"  6 = "U"
            <default>[17].<13:15> = purge   }    7 = "-"

   <default length>, INT,

      is the length, in bytes, of the <default> array; it must
      be given if <default> is given.  This number should always
      be specified as 36; in the future, new fields may be added
      to <default>, requiring <default length> 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.
   =  (CCE) indicates a successful verification and/or logon.
   >  (CCG) indicates that there is no such user, or that the
            password is invalid.

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, 36 );
   IF < THEN ... ! buffer or i/o error
   ELSE IF > THEN ... ! no such user, or bad password
   ELSE ... ! successful
```

CONSIDERATIONS

- Following a successful logon via this procedure, a process is considered to be 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.

SECTION 8

MEMORY MANAGEMENT PROCEDURES

The GUARDIAN operating system provides six basic procedures for the management of memory in extended segments. These procedures, available only on NonStop II systems, are as follows:

ALLOCATESEGMENT	allocates an extended memory segment for use by the calling process
DEALLOCATESEGMENT	deallocates an extended memory segment
DEFINEPOOL	designates a portion of a user's stack or an extended segment for use as a buffer pool
GETPOOL	obtains a block of memory from a buffer pool
PUTPOOL	returns a block of memory to a buffer pool
USESEGMENT	supplies the segment ID of an extended memory segment so that the calling process may use the segment

Several privileged procedures are also provided for advanced memory management. These procedures are described in section 8.2.

SEGMENTED MEMORY (NonStop II systems only)

The normal environment for code execution in the processor is a process. Besides a current register state, each process on the NonStop II system has a segment (possibly shared) for its code space. This segment is loaded into MAP[2,0:63] when the process runs. (For a definition of the MAPs, see the NonStop II System Description Manual.) It is allocated by the operating system, and is one of four possible code spaces. It is addressable data as logical segment #2, and if the process is currently executing in this code space, as logical segment #3 as well. If the process is executing in user mode, the segment is read-only.

In addition, each process may have a library segment (possibly shared), which is loaded into MAP[4,0:63] when the process runs. If necessary, this segment is allocated by the operating system.

A process's data space is a unique segment that is loaded into MAP[0,0:63] when the process runs. This segment is allocated automatically by the operating system. It is addressable either in the usual way as the process's data stack, or as his logical segment #0. The absolute segment number is kept in the PCB. There is one segment of this type per process.

In addition, a process may also have an extended data space in one or more segments (possibly shared). If so, the base and limit for this address space is set when the process is active. A non-privileged user can use only one data segment at a time, since only one logical extended address can be bounds checked and translated into an absolute extended address. This means that the non-privileged user cannot transfer data directly from one of these segments to another. If such a transfer is required, he must move the data into his data stack, put the required segment in use, and move the data. Note, however, that these segments can be larger than 128K bytes.

The rest of the MAP contains the extended address space cache, the system data space, the two system code spaces, and i/o buffers that are randomly allocated by i/o processes.

This process environment can be used in several ways:

1. Extended user spaces can be provided by defining one or more segments, then allowing the user to access them by setting the segment bounds and limit register to define that space when the process is executing in user mode. This allows a user to have far more than 128K bytes of addressable data space. A user process can have several of these spaces, but an explicit switch must be done between them.

2. The file system control blocks and buffers can be placed in a segment (or part of a segment). When the file system is running, the segment base and limit registers can be set to this space. The file system runs in privileged mode (so that it can access the message system), but all its memory references can be checked to see that they are within either the user's stack or the control block/buffer segment. By assigning a process a set of whole pages, the system can protect individual processes from each other. Bus transfers are made to and from this buffer segment.

3. I/O processes can run with their own data space for such items as OCB's and FCB's. They can set the segment bounds to point to the pages of the "i/o space" that they have been assigned. By assigning a process a set of whole pages, the system can protect individual processes from each other. Bus transfers are made to and from either the segment space or the process's data space, as required by the process.

The following procedures are related to segmented memory:

ALLOCATESEGMENT	allocates an extended memory segment for use by the calling process
DEALLOCATESEGMENT	deallocates an extended memory segment
USESEGMENT	supplies the segment ID of an extended memory segment so that the calling process may use the segment

SPACE MANAGEMENT WITHIN A SEGMENT

In addition to the segmented memory procedures, the GUARDIAN operating system on the NonStop II system provides a procedure allowing a process to designate a portion of its stack or a portion of an extra segment for a buffer pool. Once the pool is set up, the process may allocate and free variable-size buffers from the pool by calling system procedures.

Most programs in the system use private pools which are in their own segments. However, two pools are defined for shared access by privileged callers. They are SYSPOOL, which controls all free space in the system data space, and MAPPOOL, which controls free space that is mapped into maps 6 through 14. When space is allocated or returned to these pools, the procedures use MUTEXON/OFF to provide mutual exclusion to the pool data structures. Any data structure errors in either of these pools results in a processor halt.

The following procedures are related to memory pool management:

DEFINEPOOL	designates a portion of a user's stack or an extra segment for use as a buffer pool
GETPOOL	obtains a block of memory from a buffer pool
PUTPOOL	returns a block of memory to a buffer pool

ALLOCATESEGMENT Procedure (NonStop II systems only)

The ALLOCATESEGMENT procedure allocates an extended memory segment for use by the calling process. This procedure is available only on NonStop II systems.

The call to the ALLOCATESEGMENT procedure is:

```
<status> := ALLOCATESEGMENT ( <segment id>
            --------------- - ------------
                              , <segment size>
                              , <filename>
                              , <pin> )
                                      -
```

where

<status>, INT,

is returned 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 page table space
-7	security violation on attempt to share segment
-8	<pin> does not exist
-9	<pin> does not have the segment allocated

<segment id>, INT,

is the number by which the process wishes to refer to the segment. Segment ID's in the range of:

0-1023	may be specified by user processes
1024-2047	are reserved for Tandem-supplied software
2048-4095	may be used only by privileged processes

No process may supply a segment ID greater than 4095. Segment ID 2048 is reserved for the process file segment (PFS), and segment ID's 3072 through 4095 are assumed to be i/o segments.

→

```
    <segment size>, INT:32,

      if present, is the number of bytes that the segment must
      hold.  This value must be greater than zero and less than
      %777777777D.  If this parameter is not supplied, then the
      <pin> parameter must be supplied.

    <filename>, 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 through the end-of-file is used as initial
      data for the segment.  If the file does not exist, it
      is created, with a file size equal to the segment size.
      When the process terminates, any data still in memory
      is written back out to the file.  If the parameter is
      not specified, or a blank terminated volume is supplied,
      a temporary file is created for the segment on either
      the program file's volume or the specified volume,
      respectively.  The procedure assures that all extents for
      the file have been allocated.

      If a temporary file name is specified in the call to
      ALLOCATESEGMENT, the system will simply open the file and
      not attempt to create it.  When the segment is deallocated,
      the swapfile will be purged automatically.

    <pin>, INT,

      if present, designates that the segment specified by
      <segment id> is to be shared with the process specified
      by <pin>.  In order for this to occur, the processes
      must share the same access ID, or this process's access ID
      must be the group manager for the other's access ID, or
      this process' access ID must be the super ID.

examples:

    status := ALLOCATESEGMENT ( segment^id, seg^size, swap^file );
                  !  standard call to create a user segment;
                  !  "swap^file" parameter may be omitted

   status := ALLOCATESEGMENT ( segment^id,,, pin );
                 !  call to share the segment "segment^id"
                 !  with the process "pin"
```

DEALLOCATESEGMENT Procedure (NonStop II systems only)

The DEALLOCATESEGMENT procedure deallocates an extended memory segment when it is no longer needed by the calling process. This procedure is available only on NonStop II systems.

The call to the DEALLOCATESEGMENT procedure is:

```
CALL DEALLOCATESEGMENT ( <segment id> , <flags> )
---- ----------------- - ------------           -

where

   <segment id>, INT,

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

   <flags>, INT,

     if present, has the form:

        <0:14>   must be zero (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 zero.

example:

   CALL DEALLOCATESEGMENT ( segment^id );
```

The DEFINEPOOL procedure designates a portion of a user's stack or an extended segment for use as a buffer pool. This procedure is available only on NonStop II systems.

The call to the DEFINEPOOL procedure is:

```
<status> := DEFINEPOOL ( <pool head> , <pool> , <pool size> )
            ---------- - ---------- - ------ - ---------- -

where

   <status>, INT,

     is returned 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>, INT:ref:EXT:19,

     is a pointer to the memory space, within an allocated
     segment, to be used as the pool header.

   <pool>, INT:ref:EXT,

     is a pointer to the memory space to be used for the pool.

   <pool size>, INT:32,

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

example:

   status := DEFINEPOOL ( pool^head, pool, 2048 );
```

GETPOOL Procedure (NonStop II systems only)

The GETPOOL procedure obtains a block of memory from a buffer pool. This procedure is available only on NonStop II systems.

The call to the GETPOOL procedure is:

```
<address > := GETPOOL ( <pool head> , <block size> )
              ------- - ----------- - ------------ -

where

   <address>, INT:32,

     is returned the address of the memory block obtained if
     the operation was successful, or -1D if an error occurred
     or if <block size> was zero.

   <pool head>, INT:ref:EXT:19,

     is a pointer to a pool head previously defined by
     a call to DEFINEPOOL.

   <block size>, INT:32,

     is the size, in bytes, of the memory to be obtained from
     the pool.  This number cannot be greater than %377770D.
     To check data structures without getting any memory from
     the pool, <block size> may be set to zero.

condition code settings:

   <  (CCL)  indicates that <block size> was out of range, or
             that the data structures were invalid; -1D is
             returned.
   =  (CCE)  indicates that the operation was successful;
             extended address of block returned if <block size>
             greater than zero, or -1D returned if <block size>
             equal to zero.
   >  (CCG)  indicates that insufficient memory was available;
             -1D is returned.

examples:

   buf^ptr := GETPOOL ( pool^head, 128 );   ! returns 128 bytes
   dummy^var := GETPOOL ( pool^head, 0 );   ! checks data
                                            !    structures
```

CONSIDERATIONS

- A process that has destroyed data structures may get a bounds violation trap on a call to GETPOOL or PUTPOOL.

The PUTPOOL procedure returns a block of memory to a buffer pool. This procedure is available only on NonStop II systems.

The call to the PUTPOOL procedure is:

```
CALL PUTPOOL ( <pool head> , <address> )
---- ------- - ----------- - --------- -

where

   <pool head>, INT:ref:EXT:19,

     is a pointer to the pool head of the pool from which
     the block of memory was obtained.

   <address>, .EXT

     is the address of the block to be returned to the pool.

condition code settings:

   <  (CCL)  indicates that the data structures were invalid.
   =  (CCE)  indicates that the operation was successful.
   >  (CCG)  is not returned by PUTPOOL.

example:

   CALL PUTPOOL ( pool^head, buf^ptr );
```

CONSIDERATIONS

- A process that has destroyed data structures may get a bounds violation trap on a call to GETPOOL or PUTPOOL.

USESEGMENT Procedure (NonStop II systems only)

The USESEGMENT procedure supplies the segment ID of an extended memory segment so that the calling process may use the segment. This procedure is available only on NonStop II systems.

The call to the USESEGMENT procedure is:

```
<old segment id> := USESEGMENT ( <segment id> , <pin> )
                    ----------

where

   <old segment id>, INT,

     is returned the segment ID of the previously used segment,
     if any; otherwise, -1.

   <segment id>, INT,

     if present, is the segment ID of the segment to be used,
     or -1 if no segment is to be used.  If this parameter is
     not supplied, the current segment remains unchanged.

   <pin>, INT,

     if present, is the process number of another process
     whose segment is to be shared by the calling process.
     This parameter may be specified only by privileged
     callers.

condition code settings:

   <  (CCL)  indicates that <segment id> is not allocated, or
             that <pin> or an absolute segment was supplied
             by a non-privileged caller.
   =  (CCE)  indicates that the operation was successful.
   >  (CCG)  is not returned by USESEGMENT.

examples:

   old^seg^id := USESEGMENT ( new^seg^id );  ! change segments
   segment^id := USESEGMENT;          ! get current segment ID
   segment^id := USESEGMENT ( -1 ); ! de-select extended segments
```

CONSIDERATIONS

- Because segment relocation is done, the first byte of any extended segment has the address %2000000D.

The GUARDIAN operating system provides several procedures for advanced memory management: one procedure for NonStop systems, and two procedures for NonStop II systems. To call these procedures, a program must be executing in privileged mode.

The procedures are:

LOCKDATA (NonStop systems only)

> permits a process to make a portion of its data area main-memory resident and, optionally, causes the pages to be entered into the system data map

LOCKMEMORY (NonStop II systems only)

> permits a process to lock arbitrary buffers in arbitrary memory segments, both data and code

UNLOCKMEMORY (NonStop II systems only)

> permits a process to unlock arbitrary buffers in arbitrary memory segments, both data and code

Note: Locking code in a NonStop or NonStop II system can be accomplished only for procedures that have been assigned the RESIDENT attribute.

LOCKDATA Procedure (NonStop systems only)

The LOCKDATA procedure is used to make a block of data in the application process data area main-memory resident and, optionally, causes the pages to be entered into the system data map as required for resident buffering. A process calling LOCKDATA must be executing in privileged mode; otherwise, an instruction failure trap will occur. This procedure is available only on NonStop systems; for NonStop II systems, see the LOCKMEMORY and UNLOCKMEMORY procedures.

The call to the LOCKDATA procedure is:

```
{ <state> := } LOCKDATA ( <address> , <count> , <sys map> )
{ CALL       } -------- - --------- - ------- - --------- -
```

where

<state>, INT,

is either 1, indicating that the data was locked, or 0, indicating that the call to LOCKDATA failed.

<address>, INT:value,

is the 'G'[0] relative address of the first word in the data area to be made resident. Note that this is a value parameter and, therefore, should be passed in the form: @<variable>.

<count>, INT:value,

is the number of words in the block to be made resident (starting with <address>).

<sys map>, INT:value,

indicates whether or not the block of data should be assigned to system data map entries:

0 = do not assign to system map entries
1 = assign to system map entries

example:

```
IF NOT LOCKDATA (@buffer,256,1) THEN ...
```

CONSIDERATIONS

- Once a data area is made main-memory resident, it remains resident until the process is deleted.

- Physical pages are made resident from the location indicated by <address> to the location indicated by <address> plus <count> minus one.

- If too many pages are made resident, no memory space will be available for virtual memory. "No memory available" traps may occur.

- Assigning entries to the system data map is used when specifying resident buffers for a file.

- A process can place only one block of memory in the system map.

- If LOCKDATA is called and <sys map> = 1 is specified, a "No memory available" trap will occur if no system map entries are available.

- Space for assignment of system data map pages is obtained from system data map entries not used by the operating system. The maximum number of system data map entries available for assignment via LOCKDATA is therefore configuration and processor module dependent.

  The system data map entries available for this purpose begin following the last page used by the operating system in the lower 32K of the system data area, and end with the first page in the upper 32K used by the operating system in the system data area. (SYSGEN makes assignments in the lower 32K starting at address 0 and working upward; assignments in the upper 32K are made starting at address %177777 - the highest address - and working downward).

  The number of pages available in a given processor module is computed by using the "SYSTEM ADDRESS SPACE USED" entry on the SYSGEN listing for that processsor as follows:

  1. Take the <l limit> address shown on the SYSGEN listing and round it up to the nearest page boundary, then divide that value by the page size in words (i.e., 1024):

     <base page> := (<l limit> + 1024) / 1024

  2. Take the <u base> address shown on the SYSGEN listing and divide that value by the page size in words:

     <lim page> := <u base> / 1024

  3. Subtract the <basepage> value from the <lim page> value to obtain the number of available pages.

     <num pages> := <lim page> - <base page>

  For example, the number of system data pages available for assignment in a process module having the following "SYSTEM ADDRESS SPACE USED" values displayed is

```
    .
    .
  VIRTUAL SYSTEM DATA: 2  K
  SYSTEM ADDRESS SPACE USED: %000000-%061777 %133463-%177777
                                     ^          ^
                                     |          |
                                  <l limit>  <u base>
```

<base page> is (%061777 + %2000) / %2000 = %31 (25)

<lim page> is %133463 / %2000 = %55 (45)

<num pages> is %55 - %31 = %24 (20)

The LOCKMEMORY procedure permits a process to lock arbitrary buffers in arbitrary memory segments, both data and code. This procedure is available only on NonStop II systems; for NonStop systems, see the LOCKDATA procedure.

The call to the LOCKMEMORY procedure is:

```
<status> := LOCKMEMORY ( <address>
            ---------- - ---------
                       , <byte count>
                       - ------------
                       , <timeoutvalue>
                       - --------------
                       , <parameter1>
                       , <parameter2> )
                                      -
```

where

<status>, INT,

is returned a status word having one of the following values:

0	no error -- all pages were present initially; storage locked.
1	one or more pages were absent and have been brought in; storage locked.
2	waiting for memory; no storage locked, or not all storage locked.
-1	no storage available, or timed out waiting for memory; no storage locked.
-2	bounds violation, illegal timeout value, or missing parameter; no storage locked.

<address>, STRING:ref:EXT,

is the extended address of the block to be locked down.

<byte count>, INT(32):value,

is the number of bytes to be locked.

<timeoutvalue>, INT(32):value,

specifies the time period, in .01-second units, that the procedure is to wait for memory. This value must be zero or greater; zero specifies no waiting.

→

```
    <parameter1>, INT,

      if present, identifies the timeout message read from
      $RECEIVE.

    <parameter2>, INT(32):value,

      if present, identifies the timeout message read from
      $RECEIVE (same purpose as <parameter1>).

examples:

    status := LOCKMEMORY ( address, byte^count, 1000);
                ! lock, waiting up to 10 seconds for memory
    status := LOCKMEMORY ( address, byte^count, 1000,
                parm1, parm2 );
                ! lock, and continue processing while waiting
                ! for memory
```

CONSIDERATIONS

- If the <timeoutvalue> parameter is zero, memory will be allocated as long as any memory is available. If the <timeoutvalue> parameter is nonzero, however, memory will be allocated only if all the required amount of memory is available; otherwise, it will wait, and may eventually time out.

- If either <parameter1> or <parameter2>, or both, are supplied and not all the requested memory is immediately available, LOCKMEMORY lets the process resume and returns a <status> value of 2 (waiting for memory); at the end of the specified time period, the procedure signals the completion or failure of the lock request on $RECEIVE. If neither of the two parameters is supplied, a <status> value of -1 is returned after timeout, and a memory lock failure message is sent to $RECEIVE. The memory lock completion and memory lock failure messages have the following form:

```
<sysmsg>              =  -23 for completion
                         -24 for failure
<sysmsg>[1]           =  parameter1 supplied to LOCKMEMORY
                            (if none supplied, 0)
<sysmsg>[2] FOR 2     =  parameter2 supplied to LOCKMEMORY
                            (if none supplied, 0D)
```

- If the lock is delayed because of lack of available memory, the procedure allocates two Time List Elements (TLE's); one is added to the time list, the other to the lockwait list. Whenever a page gets unlocked and there is a TLE on the lockwait list, the memory manager is awakened with a certain event. The memory

manager then, under mutual exclusion, deletes the TLE from the time list and proceeds with the lock. If the memory manager fails again, it stores back the remaining byte count and the current address, and adds the TLE to the time list again with the remaining time value. Then the memory manager proceeds the same way on the next unlock.

- If the TLE on the time list times out, the time list interrupt handler unlocks already locked pages, releases the TLE from the lockwait list, and queues the TLE from the time list on the $RECEIVE queue for the process. A memory lock failure message is then read from $RECEIVE. If, however, enough memory becomes available during the time period and the memory manager succeeds with the lock request, a memory lock completion message is read from $RECEIVE. In either case, the READ (or AWAITIO) completes with a CCG and error #6.

- Another way for privileged users to lock and unlock memory is to allocate a resident segment using the ALLOCATESEGMENT procedure, define a pool in it with DEFINEPOOL, and use GETPOOL and PUTPOOL. GETPOOL and PUTPOOL automatically lock and unlock storage. (The ALLOCATESEGMENT, DEFINEPOOL, GETPOOL, and PUTPOOL procedures are described in section 8.1.)

UNLOCKMEMORY Procedure (NonStop II systems only)

The UNLOCKMEMORY procedure permits a process to unlock arbitrary buffers in arbitrary memory segments, both data and code. This procedure is available only on NonStop II systems; for NonStop systems, see the LOCKDATA procedure.

The call to the UNLOCKMEMORY procedure is:

```
  CALL UNLOCKMEMORY ( <address> , <byte count> )
  ---- ------------ - --------- - ------------ -

where

   <address>, STRING:ref:EXT,

     is the extended address of the block to be unlocked.

   <byte count>, INT(32):value,

     is the number of bytes to be unlocked.

example:

   CALL UNLOCKMEMORY ( address, byte^count );
```

SECTION 9

SEQUENTIAL I/O PROCEDURES

The sequential i/o procedures are a standardized set of procedures for performing common input and output operations. These operations include reading and writing IN and OUT files, and handling BREAK from a terminal. The sequential i/o procedures are intended primarily for use by Tandem subsystem and user utility programs. Programs using these procedures can treat different file types in a consistent and predictable manner.

Some characteristics of the sequential i/o procedures are:

- All file types are accessed in a uniform manner.
  - File access characteristics, such as access mode, exclusion modes, and record size, are selected according to device type and the intended access.
  - The sequential i/o procedures' default characteristics are set to facilitate their most general use.
- Error recovery is automatic. All fatal errors cause the display of a comprehensive error message, all files to close, and the process to abort. The automatic error handling and/or the display of error messages may be turned off. This allows the program to do the error handling.
- The characteristics of sequential i/o operations can be altered at open time with the OPEN^FILE procedure. This is also possible before or after the open time with the SET^FILE procedure. Some optional characteristics are:
  - Record blocking/deblocking
  - Duplicative file capability, where data read from one file is automatically echoed to another file
  - An error reporting file where all error messages are directed. When a particular file is not specified, the error reporting file is the home terminal.

Introduction

- The sequential i/o procedures can be used with the INITIALIZER procedure to make run-time changes. File transfer characteristics, such as record length, can be changed using the Command Interpreter ASSIGN command. (See "Interface with INITIALIZER and ASSIGN Messages".)

- The sequential i/o procedures retain information about the files in file control blocks. There is one File Control Block (FCB) for each open file, plus one common File Control Block which is linked to the other FCBs. (See "FCB Structure".)

The sequential i/o procedures and their functions are:

CHECK^BREAK	checks whether the BREAK key was typed
CHECK^FILE	retrieves file characteristics
CLOSE^FILE	closes a file
GIVE^BREAK	disables the BREAK key
OPEN^FILE	opens a file for access by the sequential i/o procedures
READ^FILE	reads from a file
SET^FILE	sets or alters file characteristics
TAKE^BREAK	enables the BREAK key
WAIT^FILE	waits for the completion of an outstanding i/o operation
WRITE^FILE	writes to a file

The sequential i/o procedures also contain a set of defines and literals that:

- Allocate control block space (see "OPEN^FILE").

- Specify open characteristics (see "OPEN^FILE").

- Set file transfer characteristics (see "SET^FILE").

- Check file transfer characteristics (see "CHECK^FILE").

Note that in the description of the procedure parameters, the commercial "at" symbol "@" is used to indicate the address of an object, not the object itself. For example, when specifying a file name to the SET^FILE procedure, the file name parameter should be passed as follows:

```
CALL SET^FILE ( in^file , ASSIGN^FILENAME , @buf );
```

where

@buf is the address of the array containing the name of the file to be opened.

SOURCE FILES

The source file named $SYSTEM.SYSTEM.GPLDEFS is used with the sequential i/o procedures. It provides the TAL defines for allocating control block space, for assigning open characteristics to the file, and for altering and checking the file transfer characteristics. The TAL literals for the sequential i/o 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.

CHECK^BREAK Procedure

The CHECK^BREAK procedure tests whether the BREAK key has been typed since the last CHECK^BREAK.

The call to the CHECK^BREAK procedure is:

```
<state> := CHECK^BREAK ( { <common FCB> } )
-------    --  -------- - { <file FCB>   } -

where

   <state>, INT,

     indicates whether or not the BREAK key has been typed.
     Values returned in <state> are:

       1 = BREAK key typed and BREAK is enabled.
       0 = BREAK key not typed or BREAK is disabled.

   <common FCB> or <file FCB>, INT:ref,

     identifies the file to be checked for BREAK.  <common FCB>
     is allowed for convenience.

example:

   CALL TAKE^BREAK ( out^file );
   WHILE NOT ( break := CHECK^BREAK ( out^file ) ) DO
     BEGIN
       .
       .
       CALL WRITE^FILE ( out^file , buffer , count );
     END;

   CALL GIVE^BREAK ( out^file );
```

CONSIDERATIONS

- If CR/LF on BREAK is enabled, the default case, a carriage return/line feed sequence is executed on the terminal where BREAK is typed.
- More information is available in "Terminals" subsection of section 2 under "Break Feature".

The CHECK^FILE procedure checks the file characteristics.

The call to the CHECK^FILE procedure is:

```
<retval> := CHECK^FILE ( { <common fcb> } , <operation> )
            ---------- - { <file fcb>   } - ----------- -
```

where

<retval>, INT,

is the value returned for the requested operation.

<common fcb> or <file fcb>, INT:ref,

identifies which file is checked. A common FCB can be used for certain types of checks; a common FCB must be used for the checks FILE^BREAKHIT, FILE^ERRORFILE, and FILE^TRACEBACK. Specifying an improper FCB causes an error indication.

<operation>, INT:value,

specifies which file characteristic is checked. The <operation>s and their associated <retval>s are:

<operation> = FILE^ABORT^XFERERR, (file must be open)
<retval> := <bit value>

returns: 0 if the process is not to abort upon detection of a fatal error in the file.
1 if the process is to abort.

<operation> = FILE^ASSIGNMASK1,
<retval> := <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.

<operation> = FILE^ASSIGNMASK2,
<retval> := <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.

→

```
<operation> = FILE^BLOCKBUFLEN,
<retval> := <block buffer length>

  returns a count of the number of bytes used for blocking.

<operation> = FILE^BREAKHIT,
<retval> := <state of the break hit bit>

  returns: 0  if the break hit bit is equal to zero in
              the FCB.
           1  if the break hit bit is equal to one in the
              FCB.

  The break hit bit is an internal indicator normally
  used only by the sequential i/o procedures.

  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.

<operation> = FILE^BWDLINKFCB,
<retval> := <backward link pointer>

  returns the address of the FCB pointed to by the backward
  link pointer within the FCB.  This indicates the
  linked-to FCB's that need to be checkpointed after an
  OPEN^FILE or CLOSE^FILE.

<operation> = FILE^CHECKSUM,          (file must be open)
<retval> := <checksum word>

  returns the value of the checksum word in the FCB.

<operation> = FILE^CREATED,           (file must be open)
<retval> := <state of the created bit>

  returns: 0  if a file was not created by OPEN^FILE.
           1  if a file was created by OPEN^FILE.

<operation> = FILE^COUNTXFERRED,      (file must be open)
<retval> := <count transferred>

  returns a count of the number of bytes transferred in the
  latest physical I/O operation.
                                                        →
```

```
<operation> = FILE^CRLF^BREAK,        (file must be open)
<retval> := <state of cr/lf break bit>

  returns: 0  if no carriage return/line feed sequence is
              to be issued to the terminal upon break
              detection.
           1  if this sequence is to be issued.

<operation> = FILE^DUPFILE,           (file must be open)
<retval> := @<dupfile fcb>

  returns the word address of the duplicate file FCB. A
  zero is returned if there is no duplicate file.

<operation> = FILE^ERROR,             (file must be open)
<retval> := <error>

  returns the error number of the latest error that
  occurred within the file.

<operation> = FILE^ERRORFILE,
<retval> := @<error file fcb>

  returns the word address within the FCB of the reporting
  error file.  A zero is returned if there is none.

<operation> = FILE^ERROR^ADDR,
<retval> := @<error>

  returns the word address within the FCB of where the error
  code is stored.

<operation> = FILE^FILEINFO           (file must be open)
<retval> := <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 device subtype are described in the
  "DEVICEINFO Procedure" subsection of sec. 2.  File types
  0-3 are described in the ENSCRIBE Programming Manual.
```

→

```
<operation> = FILE^FILENAME^ADDR,
<retval> := @<filename>

  returns the word address within the FCB of the physical
  file name.

<operation> = FILE^FNUM,                    (file must be open)
<retval> := <file number>

  returns the file number.

<operation> = FILE^FNUM^ADDR,
<retval> := @<file number>

  returns the word address within the FCB of the file
  number.  If the file is not open, the file number is -1.

<operation> = FILE^FWDLINKFCB,
<retval> := <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 FCB's that need to be checkpointed after an
  OPEN^FILE or CLOSE^FILE.

<operation> = FILE^LOGICALFILENAME^ADDR
<retval> := @<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}.

<operation> = FILE^LOGIOOUT,              (file must be open)
<retval> := <state of the logioout bit>

  returns:  0 to indicate there is no logical i/o
              outstanding.
            1 if a logical read is outstanding.
            2 if a logical write is outstanding.

<operation> = FILE^OPENACCESS
<retval> := <open access>
  returns the open access for the file.  See SET^FILE for
  the format.
```

→

```
<operation> = FILE^OPENEXCLUSION
<retval> := <open exclusion>
  returns the open exclusion for the file.  See SET^FILE
  for the format.

<operation> = FILE^PHYSIOOUT,          (file must be open)
<retval> := <state of the physioout bit>

  returns:  0 to indicate there is no outstanding physical
              i/o operation.
            1 if a physical i/o operation is outstanding.

<operation> = FILE^PRIEXT,
<retval> := <primary extent size>

  returns the file's primary extent size in pages.

<operation> = FILE^PRINT^ERR^MSG,     (file must be open)
<retval> := <state of print errmsg bit>

  returns:  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.

<operation> = FILE^PROMPT,            (file must be open)
<retval> := <interactive prompt character>

  returns the interactive prompt character for the file
  in <9:15>.

<operation> = FILE^RCVEOF,            (file must be open)
<retval> := <state of rcveof bit>

  returns:  0 if the user does not get an end-of-file
              (EOF) indication, when the last process [pair]
              having this process open, closes it.
            1 if the user does get an EOF indication when
              this process closes.

<operation> = FILE^RCVOPENCNT,        (file must be open)
<retval> := <$RECEIVE opener count>

  returns a count of current openers of this process {0:2}.
  At any given moment openers are limited to a single
  process [pair].

<operation> = FILE^RCVUSEROPENREPLY, (file must be open)
<retval> := <state of the rcv-user-open-reply bit>

  returns:  0 if the sequential i/o procedures are to
              reply to the open messages ($RECEIVE file).
            1 if the user is to reply to the open messages.
```

→

```
<operation> = FILE^READ^TRIM,          (file must be open)
<retval> := <state of the read trim bit>

  returns:  0 to indicate the trailing blanks are not
              trimmed off the data read from this file.
            1 if the trailing blanks are trimmed.

<operation> = FILE^RECORDLEN,
<retval> := <record length>

  returns the logical record length.

<operation> = FILE^SECEXT,
<retval> := <secondary extent size>

  returns the file's secondary extent size in pages.

<operation> = FILE^SEQNUM^ADDR
<retval> := @<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 non-edit file, this is the
  sequence number of the last record multiplied by 1000.

<operation> = FILE^SYSTEMMESSAGES,     (file must be open)
<retval> := <system message mask>

  returns a mask word indicating which system messages the
  user handles directly.  See SET^FILE for the format.
  A zero 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.

<operation> = FILE^SYSTEMMESSAGESMANY,
                                       (file must be open)
<retval> := @<system message mask words>

  returns a four-word mask indicating which system messages
  the user handles directly.  See SET^FILE for the format.
  All zeros indicates that the sequential i/o procedures
  handle all system messages.

<operation> = FILE^TRACEBACK,
<retval> := <state of traceback bit>

  returns: 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.
```

→

```
      <operation> = FILE^USERFLAG,
      <retval> := <user flag>

        returns the user flag word. (See SET^FILE procedure,
        SET^USERFLAG operation.)

      <operation> = FILE^USERFLAG^ADDR
      <retval> := @<user flag>

        returns the word address within the FCB of the user
        flag word.

      <operation> = FILE^WRITE^FOLD,        (file must be open)
      <retval> := <state of the write-fold bit>

        returns:  0 if records longer than the logical record
                    length are truncated.
                  1 if long records are folded.

      <operation> = FILE^WRITE^PAD,         (file must be open)
      <retval> := <state of write-pad bit>

        returns:  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.

      <operation> = FILE^WRITE^TRIM,        (file must be open)
      <retval> := <state of the write-trim bit>

        returns:  0 if trailing blanks are not trimmed from
                    data written to the file.
                  1 if trailing blanks are trimmed.

examples:

   INT .infile^name;
   @infile^name := CHECK^FILE ( infile , FILE^FILENAME^ADDR);

   INT .infnum;
   @infnum := CHECK^FILE ( infile , FILE^FNUM^ADDR);

   IF ( error := CHECK^FILE ( infile , FILE^ERROR ) ) THEN..
```

CONSIDERATIONS

- During the execution of this procedure, the detection of any error causes the display of an error message, and the process is aborted.

CLOSE^FILE Procedure

The CLOSE^FILE procedure is used to close a file.

The call to the CLOSE^FILE procedure is:

```
{ CALL          } CLOSE^FILE ( { <common fcb> }
{ <error> :=    } ------------ - { <file fcb>   }

                               , <tape disposition> )
```

where

<error>, INT,

is either a file management or a sequential i/o 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 zero.

<common fcb>, INT:ref,

identifies all files to be closed. If the break for any file is currently enabled, it is disabled.

<file fcb>, INT:ref,

identifies the file to be closed. If the break for the file is currently enabled, it is disabled.

<tape disposition>, INT:value,

specifies mag tape disposition,

where

<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.

example:

```
CALL CLOSE^FILE ( common^fcb );
CALL CLOSE^FILE ( rcv^file );
```

CONSIDERATIONS

- Edit files or files that are open with write access and blocking capability must be closed with the CLOSE^FILE procedure (or a WRITE with count-1) or the data may be lost.
- If break was taken, CLOSE^FILE gives break.
- For tapes with write access, SIO writes two EOF marks (control 2).
- CLOSE^FILE completes all outstanding nowait i/o on files that are to be closed.
- If the file is $RECEIVE and the user is not handling close messages SIO will wait for a message from each opener and reply with either error 45 if readonly access, or error 1 if readwrite access until there are no more openers (each opener has closed the process by calling CLOSE^FILE). See also $RECEIVE handling - CLOSE FILE.

GIVE^BREAK Procedure

The GIVE^BREAK procedure returns BREAK to the previous owner (the process that had BREAK enabled before the last call to TAKE^BREAK).

The call to the GIVE^BREAK procedure is:

```
{ CALL         }  GIVE^BREAK  (  { <common fcb> }  )
{ <error> := }  ----------  -  { <file fcb>   }  -
```

where

<error>, INT,

  is a file system or sequential i/o procedure error indicating the outcome of the operation.

<common fcb>, INT:ref, or

<file fcb>, INT:ref,

  identifies the file returning BREAK to previous owner. <common fcb> is allowed for convenience. If BREAK is not enabled, this call is ignored.

example:

```
CALL TAKE^BREAK ( out^file );
WHILE NOT ( break := CHECK^BREAK ( out^file ) ) DO
  BEGIN
    .
    .
    CALL WRITE^FILE ( out^file , buffer , count );
  END;

CALL GIVE^BREAK ( out^file );
```

The OPEN^FILE procedure permits access to a file using the other sequential i/o procedures.

The call to the OPEN^FILE procedure is:

```
{ CALL       } OPEN^FILE ( <common fcb> , <file fcb>
{ <error> := } ---------   ------------   ----------
                         , <block buffer>
                         , <block buffer length>
                         , <flags>
                         , <flags mask>
                         , <max record length>
                         , <prompt char>
                         , <error file fcb> )
                                            -
```

where

<error>, INT,

is a file management or sequential i/o procedure error number indicating the outcome of the operation.

If the abort-on-open-error mode is in effect, the only possible value of <error> is zero.

<common fcb>, INT:ref,

is an array of FCBSIZE words for use by the sequential i/o procedures. Only one common 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 zero before the first OPEN^FILE call following a process startup.

<file fcb>, INT:ref,

is an array of FCBSIZE words for use by the sequential i/o procedures. The file FCB uniquely identifies this file to the other sequential i/o procedures. The file FCB must be initialized with the name of the file to be opened before the OPEN^FILE call is made.

See "Initializing the File FCB" following the description of the FCB structure.

→

<block buffer>, INT:ref,

(optional) is an array used for record blocking and deblocking. No blocking is performed if <block buffer> or <block buffer length> is omitted, or if the <block buffer length> is insufficient according to the record length for the file, or if read/write access is indicated.

Blocking is performed when this parameter is supplied, the block buffer is of sufficient length, as indicated by the <block buffer length> parameter, and blocking is appropriate for the device.

The block buffer must be located within 'G' [0:32767] of the data area.

<block buffer length>, INT:value,

if present, 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>, 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 zero. The bit fields in <flags> are defined in appendix D. These literals may be combined using signed addition, since bit 0 is not used.

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 abends. If off, the file system or sequential i/o procedure error number is returned 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, like a call to any sequential i/o procedure except OPEN^FILE, all files are closed and the process abends. If off, the file system or the sequential i/o procedure error number is returned to the caller.

→

AUTO^CREATE,

auto create, defaults to on. If on, and open access is "write", a file is created, provided 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,

non-disc 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 is returned.

NOWAIT,

no-wait i/o, defaults to off (wait i/o). If on, no-wait i/o is in effect. If NOWAIT is specified in the open flags of OPEN^FILE, then the no-wait depth is 1. It is not possible to use a no-wait 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, or 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.

→

<flags mask>, INT(32):value,

if present, 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 one in the bit position corresponding to the <flags> parameter. A zero indicates the default setting is used. When omitted, all positions are treated as zeros.

<max record length>, INT:value,

if present, 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 record length> is not zero. If <max record length> is zero, then any record length is permitted.

<prompt char>, INT:value,

if present, 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>, 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 is the one used. Omitting this parameter does not alter the current error reporting file setting.

The error reporting file is used for reporting errors when possible. If this file cannot be used or the error is with the error reporting file, the default error reporting file is used. This 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. Note that the error file FCB must be initialized. See "Initializing the File FCB".

CONSIDERATIONS

- If AUTO^TOF is on, a top-of-form control operation is performed to the file when the file being opened is a process or a line printer and write or read/write access is specified.

- 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.

- 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.

- 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.

- SIO procedures append data to the file if access is write only and PURGE^DATA is off (default).

Example:

```
LITERAL  prompt= ">",              !prompt character
         buffer^size = 144;        !minimum edit file buffer size

INT      error,
        .common^fcb [ 0:FCBSIZE-1 ] := 0,
        .in^file [ 0:FCBSIZE-1 ] := 0,
        .in^filename [0:11 ] := [ "$VOLUME SUBVOL  FILENAME" ],
        .buffer [ 0:buffer^size >> 1 ];

INT(32)  flags := 0D,
         flags^mask := ABORT^OPENERR; !return control on error

CALL SET^FILE ( in^file , INIT^FILEFCB );
CALL SET^FILE ( in^file , ASSIGN^FILENAME, @in^filename );
IF ( error := OPEN^FILE  ( common^fcb ,
                           in^file ,
                           buffer ,
                           buffer^size ,
                           flags ,
                           flags^mask ,
                           prompt ) ) THEN
   BEGIN
   !
   ! handle open error here
   !
   END;
```

The READ^FILE procedure is used to read a file sequentially. The file must be open with read or read/write access.

The call to the READ^FILE procedure is:

```
{ CALL       } READ^FILE ( <file fcb> , <buffer> , <count read>
{ <error> := } ---------   ----------   --------   ------------
                          , <prompt count>
                          , <max read count>
                          , <no wait> )
                                      -
```

where

<error>, INT,

is a file system or sequential i/o 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, by 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 zero.

<file fcb>, INT:ref,

identifies the file to be read.

<buffer>, INT:ref,

is where the data is returned. The buffer must be located within 'G' [0:32767] process data area.

→

<count read>, INT:ref,

if present, is the count of 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. If <no wait> is zero, the <count read> parameter is required.

<prompt count>, INT:value,

if present, 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>, INT:value,

if present, 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>, INT:value,

if present, 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.

example:

```
WHILE NOT ( error := READ^FILE ( in^file , buffer ,
                                 count ) ) DO
  BEGIN
    .
    .
  END;
```

CONSIDERATIONS

- If the file is a terminal or process, a WRITEREAD operation is performed using the interactive prompt character or <prompt count> character from <buffer>.

The SET^FILE procedure alters file characteristics and checks the old value of those characteristics being altered.

The call to the SET^FILE procedure is:

```
{ CALL       } SET^FILE ( { <common fcb> } , <operation>
{ <error> := } -------- - { <file fcb>   } - -----------
                                          , <new value>
                                          , <old value> )
                                                        -
```

where

<error>, INT,

is a file system or sequential i/o 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 zero.

<common fcb>, INT:ref,

identifies those files whose characteristics are to be altered. The common FCB can be used for certain operations; it must be used for the operations SET^BREAKHIT, SET^ERRORFILE, and SET^TRACEBACK. If an improper FCB is specified, an error is indicated.

<file fcb>, INT:ref,

identifies the file whose characteristics are to be altered. If an improper FCB is specified, an error is indicated.

<operation>, INT:value,

specifies the file characteristic to be altered. See "List of SET^FILE Operations".

<new value>, INT:value,

specifies a new value for the specified <operation>. This may be optional, depending on the operation desired.

→

```
    <old value>, INT:ref,

      is a variable in which the current value for the specified
      <operation> is returned.  This can vary from 1 word to 12
      words, and is useful in saving this value for reset later.
      If <old value> is omitted, the current value is not
      returned.


LIST OF SET^FILE OPERATIONS

This is a list of the file characteristics which can be altered
by the SET^FILE procedure.  All addresses passed are assumed to
be integer addresses.

<operation> = ASSIGN^BLOCKBUFLEN (or, ASSIGN^BLOCKLENGTH)
<new value> = <new block length>   (optional; file must be closed)
<old value> := <block length>                              (optional)

  specifies the block length (in bytes) for the file.

<operation> = ASSIGN^FILECODE
<new value> = <new file code>      (optional; file must be closed)
<old value> := <file code>                                 (optional)

  specifies the file code for the file.

<operation> = ASSIGN^FILENAME
<new value> = @<file name>         (optional; file must be closed)
<old value> := <file name> FOR 12 words                    (optional)

  specifies the physical name of the file to be opened.  This
  operation is not used when the INITIALIZER procedure is called
  to initialize the File Control Blocks.

example:

    CALL SET^FILE ( in^file , ASSIGN^FILENAME , @in^filename );


<operation> = ASSIGN^LOGICALFILENAME
<new value> = @<logical file name> (optional; file must be closed)
<old value> := <logical file name> FOR 4 words             (optional)

  specifies the logical name of the file to be opened.  The
  <logical file name> must be encoded as follows:
                                                                  →
```

```
  byte numbers

  [0]  [1]                  [8]
  <len><logical file name>

  <len> is the length of the logical file name {0:7}.

<operation> = ASSIGN^OPENACCESS
<new value> = <new open access>    (optional; file must be closed)
<old value> := <open access>                              (optional)

  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.

<operation> = ASSIGN^OPENEXCLUSION
<new value> = <new open exclusion> (optional; file must be closed)
<old value> := <open exclusion>                           (optional)

  specifies the open exclusion for the file.  The following
  literals are provided for <open exclusion>:

    SHARE           (0)
    EXCLUSIVE       (1)
    PROTECTED       (3)

<operation> = ASSIGN^PRIEXT (or, ASSIGN^PRIMARYEXTENTSIZE)
<new value> = <new pri ext size>   (optional; file must be closed)
<old value> := <pri ext size>                             (optional)

  specifies the primary extent size (in units of 2048-byte blocks)
  for the file.

<operation> = ASSIGN^RECORDLEN (or, ASSIGN^RECORDLENGTH)
<new value> = <new record length>  (optional; file must be closed)
<old value> := <record length>                            (optional)

  specifies the logical record length (in bytes) for the file.
  ASSIGN^RECORDLENGTH gives the default read or write count.
  For defaults, see step 6 under "Initializing the File FCB."
```

→

```
<operation> = ASSIGN^SECEXT (or, ASSIGN^SECONDARYEXTENTSIZE)
<new value> = <new sec ext size>   (optional; file must be closed)
<old value> := <sec ext size>                              (optional)

  specifies the secondary extent size (in units of 2048-byte
  blocks) for the file.

<operation> = INIT^FILEFCB                    (file must be closed)
<new value> = must be omitted
<old value> = must be omitted

  specifies that the file FCB be initialized.  This operation
  is not used when the INITIALIZER procedure is called to
  initialize the File Control Blocks.

example:

    CALL SET^FILE ( common^fcb , INIT^FILEFCB );
    CALL SET^FILE ( in^file , INIT^FILEFCB );


<operation> = SET^ABORT^XFERERR                 (file must be open)
<new value> = <new state>                                  (optional)
<old value> := <state>                                     (optional)

  Sets/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 sequential i/o procedure except OPEN^FILE, all files
  are closed and the process abends.  If off, the file management
  or sequential i/o procedure error number is returned to the
  caller.

<operation> = SET^BREAKHIT
<new value> = <new state>                                  (optional)
<old value> := <state>                                     (optional)

  Sets/clears break-hit for the file.  This is used only if the
  user is handling BREAK independently of the sequential i/o
  procedures, or if the user has requested BREAK system messages
  via SET^SYSTEMMESSAGES or SET^SYSTEMMESSAGESMANY.

<operation> = SET^CHECKSUM
<new value> = <new checksum word>
<old value> := <checksum word in fcb>

  Sets/clears the checksum word in the FCB.  This is useful after
  modifying an FCB directly (i.e., without using the sequential
  i/o procedures).
```

→

```
  <operation> = SET^COUNTXFERRED                    (file must be open)
  <operation> = <new count>                                  (optional)
  <operation> := <count>                                     (optional)

    Sets the physical i/o count (in bytes) transferred for the file.
    This is used only if no-wait 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.

  <operation> = SET^CRLF^BREAK                      (file must be open)
  <new value> = <new state>                                  (optional)
  <old value> := <state>                                     (optional)

    Sets/clears carriage return/line feed on BREAK for the file.
    If on, a cr/lf is executed on the terminal when the BREAK
    key is typed.

  <operation> = SET^DUPFILE                         (file must be open)
  <new value> = @<new dup file fcb>                          (optional)
  <old value> := @<dup file fcb>                             (optional)

    specifies a duplicative file for the file.  This is a file
    where data read from <file fcb> is printed.  Defaults to no
    duplicative file.

  example:

      CALL SET^FILE (in^file, SET^DUPFILE, @out^file);

  <operation> = SET^EDITREAD^REPOSITION
  <new value> = must be omitted
  <old value> = must be omitted

    specifies that the following READ^FILE is to begin at the position
    set in the sequential block buffer (second through fourth words).
example:

    CALL SET^FILE (EDIT^FCB, SET^EDITREAD^REPOSITION);

  <operation> = SET^ERROR                           (file must be open)
  <new value> = <new error>                                  (optional)
  <old value> := <error>                                     (optional)

    Sets file system error code value for the file.  This is used
    only if no-wait 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.

  <operation> = SET^ERRORFILE
  <new value> = @<new error file fcb>                        (optional)
  <old value> := @<error file fcb>                           (optional)
```

→

Sets error reporting file for all files. Defaults to home terminal. If the error reporting file is not open when needed by the sequential i/o procedures, it is opened for the duration of the message printing, then closed.

```
<operation> = SET^OPENERSPID                    (file must be open)
<new value> = @<openers pid>                              (optional)
<old value> := <openers pid> FOR 4 words                  (optional)
```

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 sequential i/o 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.

```
<operation> = SET^PHYSIOOUT                     (file must be open)
<new value> = <new state>                                 (optional)
<old value> := <state>                                    (optional)
```

Sets/clears physical i/o outstanding for the file specified by <file fcb>. This is used only if no-wait i/o is in effect and the user makes the call to AWAITIO for the file.

```
<operation> = SET^PRINT^ERR^MSG                 (file must be open)
<new value> = <new state>                                 (optional)
<old value> := <state>                                    (optional)
```

Sets/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.

```
<operation> = SET^PROMPT                        (file must be open)
<new value> = <new prompt char>                           (optional)
<old value> := <prompt char>                              (optional)
```

Sets interactive prompt for the file. See the OPEN^FILE procedure.

```
<operation> = SET^RCVEOF                        (file must be open)
<new value> = <new state>                                 (optional)
<old value> := <state>                                    (optional)
```

Sets return EOF on process close for $RECEIVE file. This causes an end-of-file indication to be returned from READ^FILE when the receive open count goes from one to zero; 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.

```
<operation> = SET^RCVOPENCNT                    (file must be open)
<new value> = <new receive open count>                   (optional)
<old value> := <receive open count>                      (optional)
```

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 sequential i/o procedures is subsequently rejected by the user. See "SET^RCVUSEROPENREPLY".

```
<operation> = SET^RCVUSEROPENREPLY              (file must be open)
<new value> = <new state>                                (optional)
<old value> := <state>                                   (optional)
```

Sets user-will-reply for the $RECEIVE file. This is used if the sequential i/o procedures are to maintain the opener's directory and, therefore, limit opens to a single process [pair] while keeping the option of rejecting opens.

If <state> is one, <error> of 6 is returned from a call to READ^FILE when an open message is received and is the only current open by 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 sequential i/o procedures; no return is made from READ^FILE due to the rejected open attempt.

If <state> is zero, a return from READ^FILE is made only when data is received.

If "open message" = 1 is specified to SET^SYSTEMMESSAGES or SET^SYSTEMMESSAGESMANY, the setting of SET^RCVUSEROPENREPLY has no meaning.

An <error> of 6 is returned from READ^FILE if an open message is accepted by the sequential i/o procedures.

```
<operation> = SET^READ^TRIM                     (file must be open)
<new value> = <new state>                                (optional)
<old value> := <state>                                   (optional)
```

Sets/clears read-trailing-blank-trim for the file. If on, the <count read> parameter does not account for trailing blanks.

→

```
<operation> = SET^SYSTEMMESSAGES                    (file must be open)
<new value> = <new sys-msg mask>                              (optional)
<old value> := <sys-msg mask>                                 (optional)
```

Sets system message reception for the $RECEIVE file. Setting a bit in the <sys-msg mask> indicates that the corresponding message is to be passed back to the user. Default action is for the sequential i/o procedures to handle all system messages.

where

```
   <sys-msg-mask>.<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.

```
<operation> = SET^SYSTEMMESSAGESMANY                (file must be open)
<new value> = @<new sys-msg mask words>                       (optional)
<old value> := <sys-msg mask words>                           (optional)
```

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 be passed back to the user. Default action is for the sequential i/o procedures to handle all system messages.

→

```
where

   <sys-msg-mask>[0].<0> = unused

                        <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> = unused
                       <14> = unused
                       <15> = unused

   <sys-msg-mask>[1].<0> = unused

                        <1> = unused
                        <2> = unused
                        <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> = unused

                       <10> = unused
                       <11> = unused
                       <12> = unused

                       <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]     = all bits unused

<operation> = SET^TRACEBACK
<new value> = <new state>
<old value> := <old state>

  Sets/clears the traceback feature.  When traceback is active,
  the sequential i/o facility appends the caller's P-relative
  address to all error messages.

<operation> = SET^USERFLAG
<new value> = <new user flag>                              (optional)
<old value> := <user flag in fcb>                          (optional)

  Sets user flag for the file.  The user flag is a one-word
  value in the FCB which can be manipulated by the user to
  maintain information about calls to that file.

<operation> = SET^WRITE^FOLD                    (file must be open)
<new value> = <new state>                                  (optional)
<old value> := <state>                                     (optional)

  Sets/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.

<operation> = SET^WRITE^PAD                     (file must be open)
<new value> = <new state>                                  (optional)
<old value> := <state>                                     (optional)

  Sets/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.

<operation> = SET^WRITE^TRIM                    (file must be open)
<new value> = <new state>                                  (optional)
<old value> := <state>                                     (optional)

  Sets/clears write-trailing-blank-trim for the file.  If on,
  trailing blanks are trimmed from the output record before
  being written to the file.
```

The TAKE^BREAK procedure enables BREAK monitoring by a file.

The call to the TAKE^BREAK procedure is:

```
{ CALL        }   TAKE^BREAK ( <file fcb> )
{ <error> := }   ---------- - ---------- -
```

where

<error>, INT,

is a file system or sequential i/o procedure error indicating the outcome of the operation.

<file fcb>, INT:ref,

identifies the file for which BREAK is to be enabled. If the file is not a terminal or if BREAK is already enabled for this file, the call is ignored.

example:

```
CALL TAKE^BREAK ( out^file );
WHILE NOT ( break := CHECK^BREAK ( out^file ) ) DO
  BEGIN
    .
    .
    CALL WRITE^FILE ( out^file , buffer , count );
  END;

CALL GIVE^BREAK ( out^file );
```

WAIT^FILE Procedure

The WAIT^FILE procedure is used to wait or check for the completion of an outstanding i/o operation.

The call to the WAIT^FILE procedure is:

```
<error> := WAIT^FILE ( <file fcb> , <count read> , <time limit> )
           --------- - ----------                                -
```

where

<error>, 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 has asked for system messages via 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 due to retry

<file fcb>, INT:ref,

identifies the file for which there is an outstanding i/o operation.

<count read>, 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>, 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.

→

If <time limit> is not 0D, then a wait for completion is indicated. The time limit then specifies the maximum time, in .01-second units, that the caller waits for a completion. A time limit value of -1D indicates a willingness to wait forever.

If <time limit> is 0D, then a check for completion is indicated. 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 is returned.

If <time limit> is 0D and <error> is 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". See "Checking File Transfer Characteristics".

example 1 - wait for completion:

```
CALL READ^FILE ( in^file , buffer ,,,, 1 );
   .
   .
   .
DO error := WAIT^FILE ( in^file , count )
UNTIL error <> SIOERR^IORESTARTED;
```

example 2 - check for completion:

```
IF NOT CHECK^FILE ( recv^file , FILE^LOGIOOUT ) THEN
   CALL READ^FILE ( recv^file , recv^buf ,,,, 1 );
DO error := WAIT^FILE ( recv^file , recv^count , 0D )
UNTIL error <> SIOERR^IORESTARTED;
```

WRITE^FILE Procedure

The WRITE^FILE procedure writes a file sequentially. The file must be open with write or read/write access.

The call to the WRITE^FILE procedure is:

```
{ CALL        } WRITE^FILE ( <file fcb> , <buffer> , <write count>
{ <error> := } ----------- - ---------- - -------- - -------------
                           , <reply error code>
                           , <forms control code>
                           , < no wait> )
                                        -
```

where

<error>, INT,

is a file system or sequential i/o 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 <no wait> is not zero, the only possible value for <error> is zero, when abort-on-error is in effect.

<file fcb>, INT:ref,

identifies the file to which data is to be written.

<buffer>, INT:ref,

is the data to be written. <buffer> must be located within 'G'[0:32767] the process data area.

<write count>, 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>, INT:value,

(for $RECEIVE file only) if present, is a file management error to be returned to the requesting process by REPLY. If omitted, zero is replied.

<forms control code>, 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 management 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>, 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.

example:

```
CALL WRITE^FILE ( out^file , buffer , count );
```

ERRORS

A literal is associated with each of the sequential i/o procedures errors. These messages apply to coding errors and are considered fatal. The one exception is "no-wait i/o restarted", error SIOERR^IORESTARTED.

The sequential i/o procedure message numbers, messages, and their associated meanings are:

512 SIOERR^INVALIDPARAM
SIO procedure contains invalid parameter (all procedures). Correct parameter in error.

513 SIOERR^MISSINGFILENAME
SIO procedure is missing a file name (open error). Specify file name in procedure call.

514 SIOERR^DEVNOTSUPPORTED
SIO procedures do not support specified device type (open error). Change device type.

515 SIOERR^INVALIDACCESS
Access mode is not compatible with device type (open error). This error occurs if program opens an edit file with read or write access or with blocking specified. Change device type or access mode.

516 SIO^INVALIDBUFADDR
Buffer address is not within 'G'[0:32767] of data area (open error). Move buffer into lower memory.

517 SIOERR^INVALIDFILECODE
File code specified in ASSIGN command does not match file code of file. Change file name or file code in ASSIGN command.

518 SIOERR^BUFTOOSMALL
Specified buffer is too small (open error). For reading an edit file, allocate at least 144 bytes of buffer space. For writing an edit file, allocate at least 1024 bytes of buffer space. For blocking, allocate at least same number of bytes for buffer space as for logical record length. If error persists after increasing buffer space, directory of edit file is in error. Edit the file; editor usually can correct directory error.

519 SIOERR^INVALIDBLKLENGTH
ASSIGN block length is greater than block buffer length. Correct ASSIGN command or use larger buffer.

520 SIOERR^INVALIDRECLENGTH
Specified record length is either zero or greater than <max record length> specified in OPEN^FILE; or record length for $RECEIVE file is less than 14; or record length is greater than 254 and procedure specifies variable-length records (open error). Correct the record length.

521 SIOERR^INVALIDEDITFILE
An edit file is invalid (open error). Ensure that correct file is specified.

522 SIOERR^FILEALREADYOPEN
Program used SET^FILE for a file that should be closed or used OPEN^FILE for a file that is already open. Either close file or correct procedure call (for example, change parameters to permit operation when file is open).

523 SIOERR^EDITREADERR
An edit read error occurred (open or read error).

524 SIOERR^FILENOTOPEN
Specified file is not open (check, read, set, write, or wait error). Either open file or correct procedure call (for example, change parameters to permit operation when file is closed).

525 SIOERR^ACCESSVIOLATION
Specified access mode is not compatible with requested operation (read or write error). Change operation or access mode.

526 SIOERR^NOSTACKSPACE
Program requires temporary buffer, but stack has insufficient space. Increase run-time memory size if it is less than 32K; otherwise, move one or more non-string arrays to upper memory.

527 SIOERR^BLOCKINGREQD
Program attempted a write fold or write pad without a block buffer (write error). Supply block buffer.

528 SIOERR^EDITDIROVERFLOW
Overflow occurred in internal directory of an edit file (write error). The Edit file directory size exceeded the buffer block size declared for i/o to that file.

529 SIOERR^INVALIDEDITWRITE
Program attempted to write to an edit file after writing internal directory (write error).

530 SIOERR^INVALIDRECVWRITE
Program read $RECEIVE file, but did not follow read with write to $RECEIVE (write error). Add missing write or delete extra read.

531 SIOERR^CANTOPENRECV
SIO procedure cannot open $RECEIVE for break monitoring. User did not open $RECEIVE with OPEN^FILE procedure (CHECK^BREAK error). Open $RECEIVE with OPEN^FILE to do break monitoring while using $RECEIVE.

532 SIOERR^IORESTARTED
No-wait i/o restarted. This message is a warning, not an error. Call WAIT^FILE again to continue waiting.

533 SIOERR^INTERNAL
An internal error occurred (wait error).

534 SIOERR^CHECKSUMCOM
SIO procedure encountered error while performing checksum on common FCB (all procedures). Check program for pointer errors.

535 SIOERR^CHECKSUM
SIO procedure encountered error while performing checksum on file FCB (all procedures). Check program for pointer errors.

FCB STRUCTURE

File characteristics and procedure call information are kept in a File Control Block (FCB) within the user's data space. An FCB is associated with the opening of a file, and is passed to each sequential i/o procedure to identify that file. Additionally, there is one common FCB for each process located within the user's data space. The common FCB contains information common to all files, such as a pointer to the error reporting file.

The common FCB is initialized during the first call to OPEN^FILE following process creation. This is indicated to OPEN^FILE when the first word of the common FCB is set to zero prior to calling OPEN^FILE for the first time.

An FCB is initialized prior to calling OPEN^FILE by invoking the define INIT^FILEFCB, or by declaring the FCB using the define ALLOCATE^FCB. The name of the file to be opened must also be put into the FCB by the define ASSIGN^FILENAME.

The FCB's can be located anywhere within the user's data space. The common and file FCB's are linked together forwards and backwards as shown in figure 9-1.

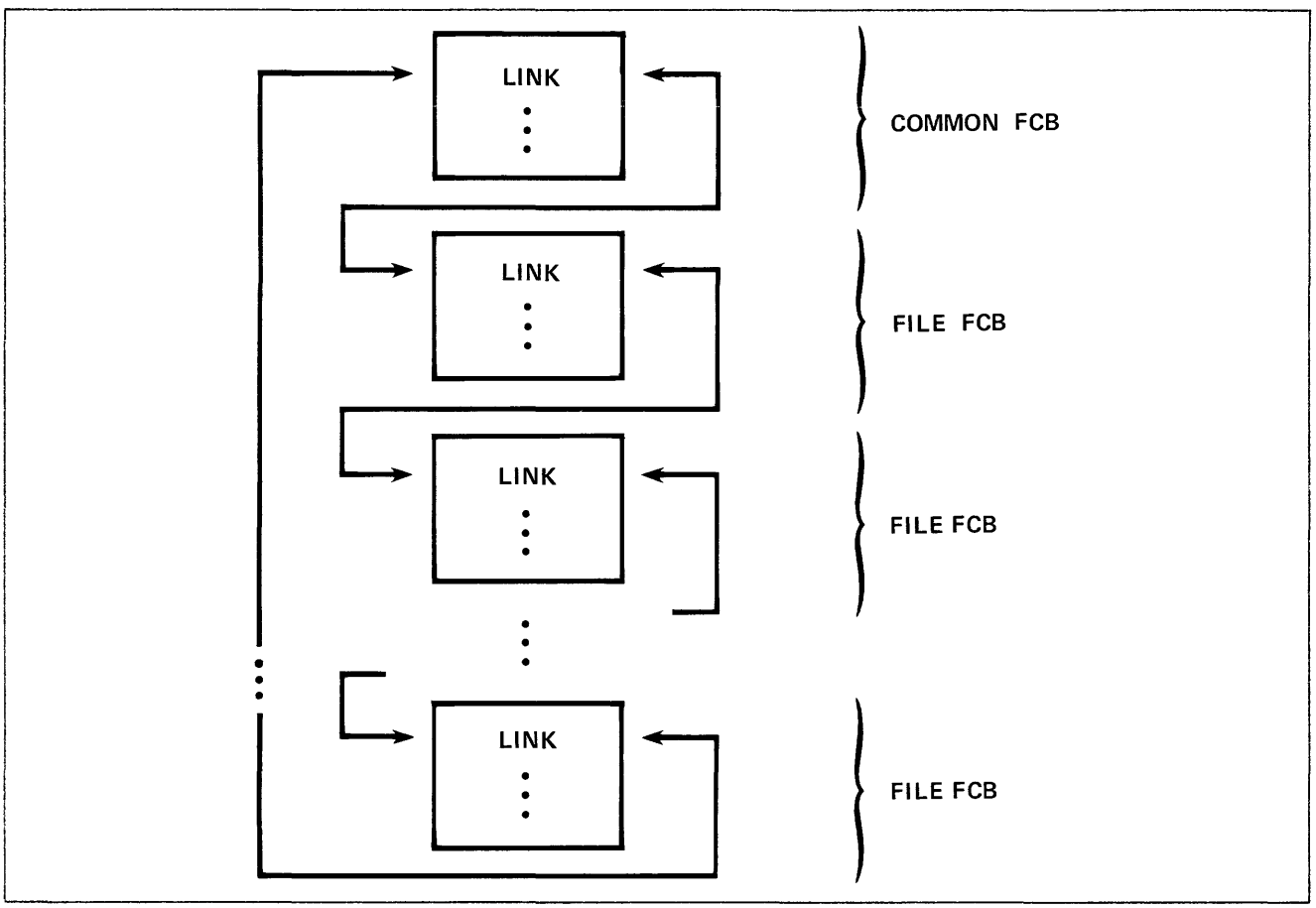


Figure 9-1. FCB Linking

Initializing the File FCB

The file FCB must be allocated and initialized before the OPEN^FILE procedure is called to open a file. The SET^FILE procedure provides these facilities, as explained in the following items.

The first three items listed - FCBSIZE, INIT^FILEFCB, and ASSIGN^FILENAME - are not used when the INITIALIZER procedure is called to initialize the file control blocks. See the INITIALIZER procedure.

1. The size in words of an FCB is provided as a literal,

   FCBSIZE (currently 60)

   example:

   ```
   INT .infile [ 0:FCBSIZE-1 ];
   ```

2. Initialize the FCB using the SET^FILE procedure. This step is required.

   ```
   CALL SET^FILE ( <file fcb> , INIT^FILEFCB )
   ---- -------- - ---------- - ------------ -
   ```

   example:

   ```
   CALL SET^FILE ( infile , INIT^FILEFCB )
   ```

3. Specify the name of the file to open. This step is required.

   ```
   CALL SET^FILE ( <file fcb> , ASSIGN^FILENAME , <file name addr> )
   ---- -------- - ---------- - -------------- - ---------------- -
   ```

   example:

   ```
   CALL SET^FILE ( in^file, ASSIGN^FILENAME , @in^filename );
   ```

4. Specify the access mode for this open. This step is optional.

   ```
   CALL SET^FILE ( <file fcb> , ASSIGN^OPENACCESS , <open access> )
   ---- -------- - ---------- - ----------------- - ------------ -
   ```

   The following literals are provided for <open access> :

   ```
   READWRITE^ACCESS (0)
   READ^ACCESS      (1)
   WRITE^ACCESS     (2)
   ```

If omitted, the access mode for the device being opened defaults to the following:

Device	Access
Operator	Write
Process	Read/Write
$RECEIVE	Read/Write
Disc	Read/Write
Terminal	Read/Write
Printer	Write
Mag Tape	Read/Write
Card Reader	Read

example:

```
CALL SET^FILE ( in^file , ASSIGN^OPENACCESS , READ^ACCESS );
```

5. Specify exclusion for this open. This step is optional.

```
CALL SET^FILE ( <file fcb> , ASSIGN^OPENEXCLUSION ,
                <open exclusion> )
```

The following literals are provided for <open exclusion> :

```
SHARE     (0)
EXCLUSIVE (1)
PROTECTED (3)
```

If omitted, the exclusion mode applied to the open defaults to the following:

Access	Exclusion Mode
Read	if terminal then share, else protected
Write	if terminal then share, else exclusive
Read/Write	if terminal then share, else exclusive

example:

```
CALL SET^FILE ( in^file , ASSIGN^OPENEXCLUSION , EXCLUSIVE );
```

6. Specify the logical record length. This step is optional.

```
CALL SET^FILE ( <file fcb> , ASSIGN^RECORDLENGTH ,
                <record length> )
```

The <record length> is given in bytes.

If omitted, <record length> defaults according to the device as follows:

Device	Logical Record Length
Operator	132 bytes
Process	132 bytes
$RECEIVE	132 bytes
Unstructured Disc	132 bytes
Structured Disc	record length defined at creation
Terminal	132 bytes
Printer	132 bytes
Mag Tape	132 bytes
Card Reader	132 bytes

7. Set the file code. This step is optional and has two meanings: 1) if AUTO^CREATE is on, the file code specifies the type of file to be created. 2) implies the file code must match the file code specified for the open to succeed.

```
CALL SET^FILE ( <file fcb> , ASSIGN^FILECODE , <file code> )
---- -------- - ---------- - -------------- - ----------- -
```

8. Set the primary extent size. This step is optional, and has meaning only if AUTO^CREATE is on.

```
CALL SET^FILE ( <file fcb> , ASSIGN^PRIMARYEXTENTSIZE ,
---- -------- - ---------- - ------------------------ -
                <primary extent size> )
                --------------------- -
```

<primary extent size> is given in pages (2048-byte units).

9. Set the secondary extent size. This step is optional, and has meaning only if AUTO^CREATE is on.

```
CALL SET^FILE ( <file fcb> , ASSIGN^SECONDARYEXTENTSIZE ,
---- -------- - ---------- - -------------------------- -
                <secondary extent size> )
                ----------------------- -
```

<secondary extent size> is given in pages, 2048-byte units.

10. Set the file's physical block length. This step is optional. The physical block length is the number of bytes transferred between the file and the process in a single i/o operation. If <block length> is specified, blocking is also specified. A physical block is composed of <block length> divided by <record length> logical records. When <block length> is not exactly divisible by <record length>, the portion of that block following the last logical record is filled with blanks.

    Note that the specified form of blocking differs from the type of blocking performed when no <block length> is specified. In the unspecified form, there is no indication of a physical block size; the records are contiguous on the medium.

    CALL SET^FILE (<file fcb> , ASSIGN^BLOCKLENGTH , <block length>)
    ---- -------- - ---------- - ------------------ - -------------- -

    <block length> is given in bytes.

INTERFACE WITH INITIALIZER AND ASSIGN MESSAGES

The sequential i/o procedures and the INITIALIZER procedure can be used in conjunction with or separately from each other. File transfer characteristics, such as record length, can be altered at run time using Command Interpreter ASSIGN commands. See the INITIALIZER procedure, section 4.

The INITIALIZER procedure reads the startup message and, optionally, the assign and param messages, from $RECEIVE. The INITIALIZER procedure can prepare global tables of a predefined structure and properly initialize FCB's with the information read from the startup and assign messages.

To use the INITIALIZER, an array called a Run-Unit Control Block must be declared. Each file to be prepared by the INITIALIZER must be initialized with a default physical file name and, optionally, with a logical file name before invoking the INITIALIZER.

The INITIALIZER reads the startup message, then requests the assign messages. For each assign message, the FCB's are searched for a logical file name which matches the logical file name contained in the assign message. If a match is found, the information from the assign message is put into the FCB. See section 11, "Command Interpreter/ Application Interface", for a description of the ASSIGN command.

The INITIALIZER also substitutes the real file names for default physical file names in the FCB's. This function provides the capability to define the IN and OUT files of the startup message as physical files and to define the home terminal as a physical file.

After invoking the INITIALIZER, the sequential i/o OPEN^FILE procedure is called once for each file to be opened.

INITIALIZER-RELATED DEFINES

Two defines are provided for allocating Run-Unit Control Block Space (CBS) and for allocating FCB space. These defines are:

1. Allocate Run-Unit Control Block and Common FCB (data declaration).

```
ALLOCATE^CBS ( <run-unit control block> , <common fcb> ,
               <numfiles> );
```

where

<run-unit control block>

is the name to be given to the run-unit control block; this name is passed to the INITIALIZER procedure.

<common fcb>

is the name to be given to the common FCB; this name is passed to the OPEN^FILE procedure.

<numfiles>

is the number of FCB's to be prepared by the INITIALIZER procedure. The INITIALIZER begins with the first FCB following ALLOCATE^CBS.

example:

```
ALLOCATE^CBS ( rucb , commfcb , 2 );
```

2. Allocate FCB (data declaration).

Note: The FCB allocation defines must immediately follow the ALLOCATE^CBS define. No intervening variables are allowed.

ALLOCATE^FCB (<file fcb> , <default physical file name>)

where

<file fcb>

is the name to be given the FCB. The name references the file in other sequential i/o procedure calls.

<default physical file name>, literal STRING,

is the name of the file to be opened. This can be an internal form of a file name or one of the following, and must be in upper case as shown.

byte numbers

```
 [0]      [8]      [16]     [24]
"         #IN               "
```

This means substitute the INFILE name of the startup message for this name.

```
"         #OUT              "
```

This means substitute the OUTFILE name of the startup message for this name.

```
"         #TERM             "
```

This means substitute the home terminal name for this name.

```
            "        #TEMP           "
```

This means substitute a name appropriate for creating a temporary file for this name.

```
            "                        "
```

This means substitute a name appropriate for creating a temporary file for this name.

If the $<volume name> or <subvol name> is omitted, the corresponding default name from the startup message is substituted for the disc file names.

example:

```
   ALLOCATE^FCB ( in^file  , "        #IN             " );
   ALLOCATE^FCB ( out^file , "        #OUT            " );
```

The following SET^FILE operation, ASSIGN^LOGICALFILENAME, is used with the INITIALIZER. The logical file name is the means by which the INITIALIZER matches an assign message to a physical file.

```
CALL SET^FILE ( <file fcb> , ASSIGN^LOGICALFILENAME ,
---- -------- - ---------- - --------------------- -
               @<logical file name> )
               -------------------- -
```

where

<file fcb>, INT:ref,

references the file to be assigned a logical file name.

@<logical file name>, INT:value,

is the word address of an array containing the logical file name. A logical file name consists of a maximum of seven alphanumeric characters, the first of which must be an alphabetic character.

<logical file name> must be encoded as follows:

byte numbers

```
[0]  [1]                 [8]
<len><logical file name>
```

<len> is the length of the logical file name.

By convention, the logical file name of the input file of the startup message should be named "INPUT"; the logical file name of the output file of the startup message should be named "OUTPUT".

example:

```
INT .buf [ 0:11 ];
STRING .sbuf := @buf '<<' 1;
sbuf ':=' [ 5, "INPUT" ];
CALL SET^FILE (  in^file , ASSIGN^LOGICALFILENAME , @buf );
sbuf ':=' [ 6, "OUTPUT" ];
CALL SET^FILE (  out^file , ASSIGN^LOGICALFILENAME , @buf );
```

Figure 9-2 shows the file assignment in relation to when the INITIALIZER is invoked. File characteristics can be set by the INITIALIZER with the ASSIGN command, or with programmatic calls to the SET^FILE procedure.

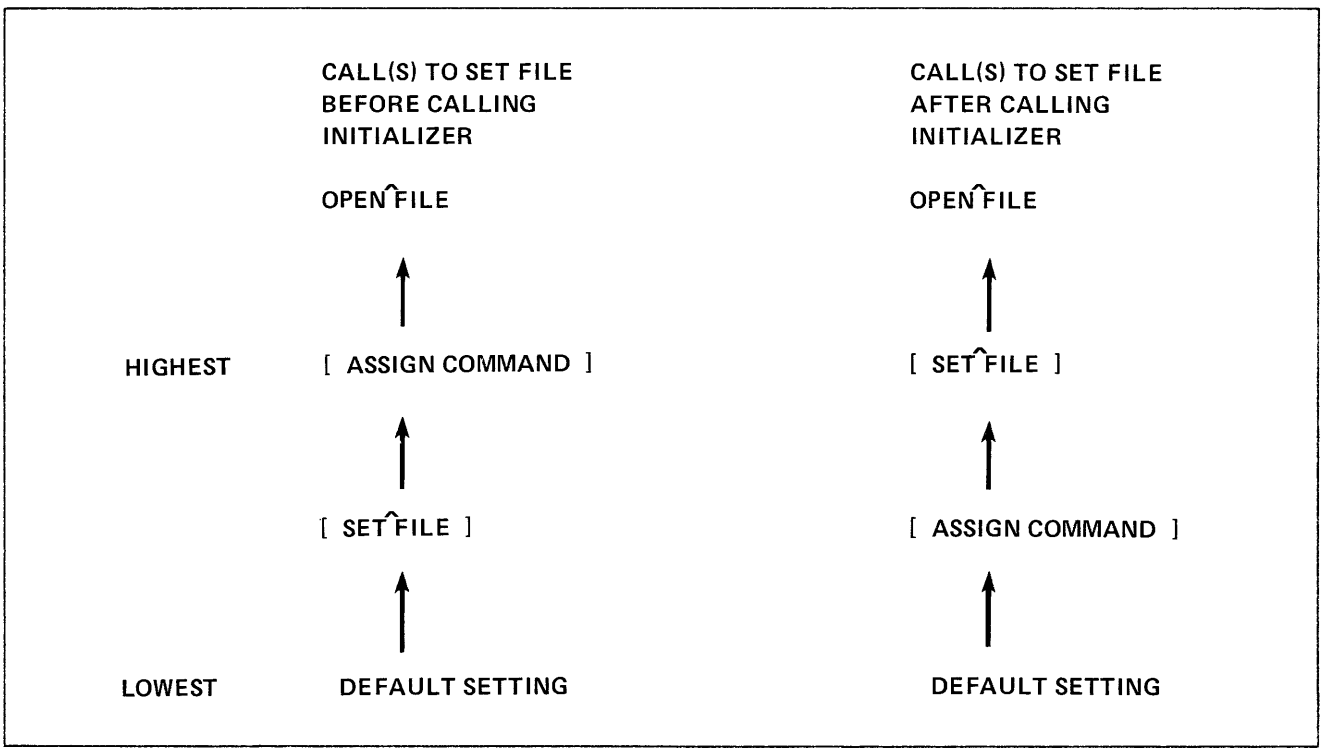


Figure 9-2. Precedence of Setting File Characteristics

CONSIDERATIONS

- If run-time changes to file transfer characteristics are not allowed then do not assign a logical file name to the file.
- In some cases it is undesirable to have the INITIALIZER assign a physical file name for the <default physical file name>. For example, when it is not desirable to default the file name, but instead to force the use of an ASSIGN command to specify a physical file for the logical file, then declare the FCB as

follows (the FCB must be adjacent to other FCB's searched by the INITIALIZER):

```
INT .<file fcb> [ 0:FCBSIZE - 1 ];
```

In the executable part of the program, before calling the INITIALIZER, initialize the FCB:

```
CALL SET^FILE ( <file fcb> , INIT^FILEFCB );
```

Assign a logical file name, and any other open attributes desired, before calling the INITIALIZER:

```
CALL SET^FILE ( <file fcb> , ASSIGN^LOGICALFILENAME , @name );
   .
CALL INITIALIZER ( .. );

CALL OPEN^FILE ( <common fcb> , <file fcb> , ... );
```

If the user neglects to ASSIGN a physical file to the logical file, the open fails with an error number 513, SIOERR^MISSINGFILENAME, "file name not supplied".

USAGE EXAMPLE

The following shows the use of the INITIALIZER and sequential i/o procedures for opening the IN and OUT files of a typical Tandem subsystem program.

If the IN and OUT files are the same file and either is a terminal or a process, only the IN file is opened. The address of the in^file FCB is put into the pointer to the out^file FCB.

The open access is assigned after the INITIALIZER is called. This overrides the open access specified in an ASSIGN command.

```
?NOLIST, SOURCE $SYSTEM.SYSTEM.GPLDEFS
?LIST
! Set up the control blocks for the INITIALIZER with supplied
! Defines.
! Initialize Run Unit Control Block and common FCB.
!     rucb     - Array holding Run Unit Control Block.
!     commfcb  - Array for the common File Control Block.

    ALLOCATE^CBS ( rucb, commfcb, 2 );

! Initialize in file FCB.
!     in^file  - Array for FCB of the in file.

    ALLOCATE^FCB ( in^file, "        #IN             " );

! Initialize out file FCB.
!     out^file - Array for FCB of the out file.
```

```
    ALLOCATE^FCB ( out^file, "        #OUT            " );

LITERAL
     process   =    0,                ! Process device type.
     terminal  =    6,                ! Terminal device type.
     inblklen  = 4096,                ! Length of block buffer for in
                                      ! file.
     outblklen = 4096,                ! Length of block buffer for out
                                      ! file.
     rec^len   =  255;                ! Maximum record length to read
                                      ! or write.

INT .inblkbuf [ 0:inblklen/2 - 1 ],   ! In file's buffer for blocking.
    .outblkbuf [ 0:outblklen/2 - 1 ],! Out file's buffer for blocking.
    .infname,                         ! In file's file name.
    .outfname,                        ! Out file's file name.
     device^type,                     ! Device type (see DEVICEINFO
                                      ! procedure, sec. 2).
     phys^rec^len,                    ! Physical record length of
                                      ! device.
     interactive;                     ! Indicates if in and out file
                                      ! are interactive, implying use
                                      ! read/write access.

INT .buf [ 0:11 ];                    ! Holds logical file names.
STRING
    .sbuf := @buf '<<' 1;             ! String corresponding to buf.

?NOLIST, SOURCE $SYSTEM.SYSTEM.EXTDECS
?LIST

PROC main^proc MAIN;
  BEGIN
    int .buffer [ 0:rec^len/2 - 1 ], ! Buffer for i/o with a single
                                      ! record.
         count := rec^len;            ! Number of bytes read in or
                                      ! written out.

!     Beginning of program execution.

!     Set up in and out files using startup message from RUN command.
        sbuf ':=' [ 5, "INPUT" ];
        CALL SET^FILE( in^file, ASSIGN^LOGICALFILENAME, @buf );
        sbuf ':=' [ 6, "OUTPUT" ];
        CALL SET^FILE( out^file, ASSIGN^LOGICALFILENAME, @buf );
        CALL INITIALIZER( rucb );

!         get physical file names for in and out files.

          @infname  := CHECK^FILE( in^file,  FILE^FILENAME^ADDR );
          @outfname := CHECK^FILE( out^file, FILE^FILENAME^ADDR );

!     Determine type of access for in file.
```

```
        CALL DEVICEINFO ( infname, device^type, phys^rec^len );
        interactive :=
          IF ( device^type.<4:9> = terminal  OR
               device^type.<4:9> = process )
             AND infname = outfname FOR 12
             THEN 1 ELSE 0;

        CALL SET^FILE( in^file, ASSIGN^OPENACCESS,
                       IF interactive THEN READWRITE^ACCESS
                  (                   ELSE READ^ACCESS );

!     Open in file.

        CALL OPEN^FILE( commfcb, in^file, inblkbuf
                        ,inblklen,,,,, out^file );

    IF interactive THEN       ! Make in and out files the same;
                              !   no need to
      @out^file := @in^file   !   open out file.
    ELSE                      ! Open out file.
      BEGIN
        CALL SET^FILE( out^file, ASSIGN^OPENACCESS, WRITE^ACCESS );
        CALL OPEN^FILE( commfcb, out^file, outblkbuf, outblklen );

!         non-interactive use, so echo reads to out file.

        CALL SET^FILE( in^file, SET^DUPFILE, @out^file );
      END;

!     Main processing loop.

!     WHILE not EOF process the record.

    WHILE ( READ^FILE( in^file, buffer, count) ) <> 1 DO
      BEGIN

!
!         Process record read in, and format a record for output.
!           .
!           .

        CALL WRITE^FILE( out^file, buffer, count );
      END;

   CALL CLOSE^FILE( commfcb );    ! close all files

  END;                            ! of main^proc
```

To change the record length of the input file, the following ASSIGN command can be entered before the program is run:

```
ASSIGN INPUT,,REC 80
```

To change the file code of the output file, the following ASSIGN command can be entered before the program is run:

```
ASSIGN OUTPUT,,CODE 9876
```

SUMMARY

The following are the steps involved to use the INITIALIZER with the sequential i/o procedures:

- Allocate the CBS and FCB, and assign the default physical file names using ALLOCATE^CBS and ALLOCATE^FCB's.
- Assign a logical file name using the SET^FILE operation, ASSIGN^LOGICALFILENAME.
- If ASSIGN command characteristics are to override program calls to SET^FILE, invoke assignment defines.
- Invoke the INITIALIZER to read the startup, assign, and param messages and prepare the file FCB's.
- If programmatic calls to SET^FILE are to override ASSIGN command characteristics, invoke assignment defines.
- Open the files with calls to OPEN^FILE.

USAGE EXAMPLE WITHOUT INITIALIZER PROCEDURE

The following example shows the use of the sequential i/o procedures for the IN and OUT files of a typical Tandem subsystem program when the INITIALIZER procedure is not used.

```
?SOURCE $SYSTEM.SYSTEM.GPLDEFS ( ... )
INT  interactive,
     error,
    .common^fcb [0:FCBSIZE-1] := 0,
    .rcv^file   [0:FCBSIZE-1],
    .in^file    [0:FCBSIZE-1],
    .out^file   [0:FCBSIZE-1],
    .buffer     [0:99],
     mompid     [0:3],
     devtype,
     junk;

LITERAL
     process       =     0,
     terminal      =     6,
     in^blkbuflen  =  1024,
     out^blkbuflen =  1024;

INT .in^blkbuf  [0:in^blkbuflen/2  - 1],
    .out^blkbuf [0:out^blkbuflen/2 - 1];

?SOURCE $SYSTEM.SYSTEM.EXTDECS ( ... )
!
! read the startup message.
!
! - open $RECEIVE.
!
CALL SET^FILE ( rcv^file , INIT^FILEFCB );
buffer ':=' "$RECEIVE  " & buffer [ 4 ] FOR 7;
! file name.
CALL SET^FILE ( rcv^file , ASSIGN^FILENAME , @buffer );
! number of bytes to read.
CALL SET^FILE ( rcv^file , ASSIGN^RECORDLENGTH , 200 );
CALL OPEN^FILE ( common^fcb , rcv^file ,,, nowait , nowait );
!
! - get mom's process ID.
!
! - first, see if I'm named.
!
CALL GETCRTPID ( MYPID , buffer );
IF buffer.<0:1> = 2 THEN
   ! not named.
   CALL MOM ( mompid );
ELSE
```

```
  BEGIN
    ! named.
    CALL LOOKUPPROCESSNAME ( buffer );
    mompid ':=' buffer [ 5 ] FOR 4;
  END;
! - allow startup message from mom only.
CALL SET^FILE ( rcv^file , SET^OPENERSPID , @mompid );
!
DO
  BEGIN
    CALL READ^FILE ( rcv^file , buffer ,,,, 1 );
    DO error := WAIT^FILE ( rcv^file , length , 3000D )
    UNTIL error <> SIOERR^IORESTARTED;
  END
UNTIL buffer = -1; ! startup message read.

! - close $RECEIVE.
CALL CLOSE^FILE ( rcv^file );
!
! see if program is being run interactively.
!
CALL DEVICEINFO ( buffer [ 9 ] , devtype , junk );
interactive :=
   IF ( devtype.<4:9> = terminal OR
        devtype.<4:9> = process  ) AND
        buffer [ 9 ] = buffer [ 21 ] for 12 THEN 1
                                           ELSE 0;

CALL SET^FILE ( in^file , INIT^FILEFCB );
CALL SET^FILE ( in^file , ASSIGN^FILENAME , @buffer [ 9 ] );
CALL SET^FILE ( in^file , ASSIGN^OPENACCESS ,
                IF interactive THEN READWRITE^ACCESS
                               ELSE READ^ACCESS );
CALL OPEN^FILE ( common^fcb , in^file , in^blkbuf , in^blkbuflen
                 ,,,,, out^file );

IF interactive THEN
   ! use in file as out file.
   @out^file := @in^file
ELSE
  BEGIN
    CALL SET^FILE ( out^file , INIT^FILEFCB );
    CALL SET^FILE ( out^file , ASSIGN^FILENAME , @buffer [ 21 ] );
    CALL SET^FILE ( out^file , ASSIGN^OPENACCESS , WRITE^ACCESS );
    CALL OPEN^FILE ( common^fcb , out^file , out^blkbuf ,
                     out^blkbuflen );
    ! set duplicative file.
    CALL SET^FILE ( in^file , SET^DUPFILE , @out^file );
  END;
.
.
.
```

Error handling and retries are implemented within the sequential i/o procedure environment by the NO^ERROR procedure. NO^ERROR is called internally by the sequential i/o procedures. If the file is opened by OPEN^FILE, then the NO^ERROR procedure can be called directly for the file system procedures.

The call to the NO^ERROR procedure is:

```
{ CALL           } NO^ERROR ( <state> , <file fcb> ,
{ <no retry> := } ---------   -------   ----------  -
                               <good error list> , <retryable> )
                               -----------------   -----------  -
```

where

<no retry>, 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 zero, one of the following is indicated:

- <state> is not zero.
- no error occurred; error is zero.
- 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>, INT:value,

if non-zero, indicates the operation is to be considered successful. The file error and retry count variables are set to zero, with <no retry> returned as non-zero. Typically, either of two values is passed in this position:

= CCE for example, immediately following 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 zero, NO^ERROR calls FILEINFO for the file.

→

<file fcb>, INT:ref,

identifies the file to be checked.

<good error list>, INT:ref,

is a list of error numbers; if one of the numbers matches the current error, <no retry> is returned as non-zero (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>, 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	n.a.
$RECEIVE	n.a.
Disc	(opened with sync depth of 1, so n.a.)
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.

example:

```
INT good^error [ 0:1 ] := [ 1, 11 ]; ! nonexistent record.

CALL SET^FILE ( out^file, SET^ERROR, 0)
DO CALL READUPDATE ( out^fnum, buffer , count )
UNTIL NO^ERROR ( = , out^file , good^error , 0 );
```

ERROR HANDLING BY NO^ERROR

Errors are handled as follows:

```
if <state> then
  begin
    fcb^error := 0;
    retrycount := 0;
    return no-retry indication
  end;

if not fcb^error then
   CALL FILEINFO ( fcb^fnum , fcb^error );
```

fcb^error	Disposition
0	return no-retry indication
1,6	READ^FILE: return no-retry indication
7	WRITE^FILE: return no-retry indication
<good^error>	return no-retry indication
100:102	prompt then if "S[TOP]" then fatal else return retry indication

110:111	if device = breakdevice then
	begin
	breakflush := 1;
	if (breakhit :=
	checkbreak
	begin
	check $receive for break message.
	if break message then
	breaktyped := 1
	else
	if breakflush then
	begin
	take break
	delay 2 sec
	end
	return breaktyped.
	end) then return no-retry indication.
	end
	delay 2 sec
	return retry indication
	end

112	begin delay 2 sec return retry indication end

```
200:201         if ( retrycount := retrycount + 1 ) > 1 then
                   goto fatal
                else return retry indication.

120, 190        if not retryable or
202:231            (retrycount := retrycount + 1) > 1 then
                   goto fatal
                else
                if device <> mag tape    and
                   device <> card reader then
                      return retry indication
                else
                   goto fatal

other           fatal:
                  if print error then
                     print an error message;
                  if abort then
                    begin
                      call close^file ( common^fcb );
                      call abend;
                    end;
                  return no-retry indication;
```

The retry count is used to determine the number of times an operation is consecutively retried for a maximum of two retries. The count is cleared when a no-retry is indicated.

$RECEIVE HANDLING

Within the environment of the sequential i/o procedures, the $RECEIVE file has two functions:

- To check for break messages
- To transfer data between processes

Within the sequential i/o procedures, these functions can be performed concurrently. It may be desirable to manage the $RECEIVE file independently of the sequential i/o procedures, and to monitor break using the sequential i/o procedures. Therefore, the SET^FILE operation SET^BREAKHIT enables the user's $RECEIVE handler to pass the break information into the sequential i/o procedure environment.

The FCB internal structure is shown in Appendix E.

$RECEIVE Data Transfer Protocol

```
RS  = RECEIVE^STATE: 0 = NEED READUPDATE, 1 = NEED REPLY.
ROC = RECEIVE^OPENER^COUNT.

  OPEN^FILE
     RS := ROC := 0;

  READ^FILE (file must be open with read or read/write access)

     if system message then
       begin
         RS := 1
         if user wants to process this message then
            return 1;
         replycode := 0
         if cpu down message then
           begin
             if cpu = opener's cpu then
                { delete process from opener's directory }
           end
         else
         if break^message then
           begin
             breakhit := 1
           end
         else
         if open^message then
           begin
             if nowait depth > 1 then replycode := 2
             else
             if ROC = 2 then
                replycode := 12
             else
```

```
      if primary open then
        begin
          if not primary pid or
             opener = primary pid then
            begin
              add primary pid to opener directory
              ROC := ROC + 1
            end
          else replycode := 12
        end
      else
      if backup open and
         ( pid in message = primary openers pid or
           not primary pid ) then
        begin
          if primary pid then
            add backup pid to opener directory
          else
            ! treat as primary open.
            add primary pid to opener directory
          ROC := ROC + 1
        end
      else replycode := 12
    end
  else
  if close message then
    begin
      if pid = primary pid then
        begin
          primary pid := backup pid
          delete backup pid from opener directory
        end
      else
      if pid = backup pid then
         delete backup pid from opener directory
      if not ( ROC := ROC - 1 ) and
         rcveof then
         error := 1
    end.

  if open message and
     user wants to reply
     and not replycode then return 1
  else
    begin
      REPLY ( replycode )
      RS := 0
    end

  return if  error = 1 then 0 else 1
end ) then return.
```

```
     if RS then REPLY ( no text, REPLYERROR = 0 ); RS := 0;

        Note: REPLY is skipped if READ^FILE immediately follows
              open.

     READUPDATE ( text ); RS := 1;
     error := 0;

  WRITE^FILE (file must be open with write or read/write access)

     if not RS then ! invalid operation
        error := SIOERR^INVALIDRCVWRITE
        RETURN;
     REPLY ( text, reply code ); RS := 0;
     error := 0;

  CLOSE^FILE

     replycode := IF access = write THEN 1 ELSE 45;
     if not RS then READUPDATE ( no text ); RS := 1;
     REPLY ( no text, replycode ); RS := 0;

     READUPDATE/REPLY until close message; RS := 0;
```

Note: To determine whether the data returned from READ^FILE is listing text or command prompt text call the file system RECEIVEINFO procedure.

NO-WAIT I/O

If NOWAIT is specified at open time, the file is opened with a no-wait i/o depth of one. Whether an individual operation is to be waited for is determined on a call by call basis. No-wait operations are completed by a call to WAIT^FILE.

If it is desirable to wait for any file, the user can call AWAITIO before calling WAIT^FILE. Depending on whether blocking is performed, a physical i/o operation may not always take place with a logical i/o operation. Therefore, the CHECK^FILE operation FILE^PHYSIOOUT is used to determine if an i/o operation is outstanding. The SET^FILE operations SET^PHYSIOOUT, SET^ERROR, and SET^COUNTXFERRED are provided to condition the FCB if the i/o is completed. The user must call WAIT^FILE following the call to AWAITIO for the file state information to be updated.

Example:

```
INT .in^fnum;

@in^fnum := CHECK^FILE ( in^file , FILE^FNUM );
error := 0;
WHILE 1 DO
  BEGIN
    IF error <> SIOERR^IORESTARTED THEN
       CALL READ^FILE ( in^file , buffer , , , 1 ); ! no wait.
      .
      .
    fnum := -1;
    CALL AWAITIO ( fnum ,, countread ,, 3000D );
    IF fnum = in^fnum THEN
      BEGIN
        CALL FILEINFO ( in^fnum , error );
        ! set i/o done.
        CALL SET^FILE ( in^file , SET^PHYSIOOUT , 0 );
        ! set count read.
        CALL SET^FILE ( in^file , SET^COUNTXFERRED , countread );
        ! set error code.
        CALL SET^FILE ( in^file , SET^ERROR , error );
        IF ( error :=
           WAIT^FILE ( in^file , in^file^countread ) ) <>
              SIOERR^IORESTARTED THEN
          BEGIN ! completed.
            !
            ! process read.
            !
          END;
      END
    ELSE
      .
      .
  END; ! WHILE 1 LOOP.
```

SUMMARY OF FCB ATTRIBUTES

The following table summarizes the operations of SET^FILE and CHECK^FILE. They are listed alphabetically by their SET^FILE name and by their CHECK^FILE name.

The following symbols are used:

FCB
- : FCB must be that of a file (not common FCB).
- C : FCB must be the common FCB.
- R : FCB must be that of the file $RECEIVE.
- A : FCB can be that of a file or common FCB (Any FCB).

State
- o : File must be open.
- C : File must be closed.
- A : File can be open or closed (Any state).

Reset
- + : These attributes have default values assigned by OPEN^FILE unless specified differently in the OPEN^FILE flags or by a later SET^FILE operation.
- - : When a file is closed these attributes are cleared. If the FCB is reopened, these attributes have default values assigned by OPEN^FILE unless specified differently in the OPEN^FILE flags or by a SET^FILE operation.

Addr
- * : The parameter passed to SET^FILE is the address of the array, i.e . @file^name instead of file^name. The value returned by CHECK^FILE is the address of the item.

F C B	S t a t e	R e s e t	SET^FILE	F C B	S t a d d r	CHECK^FILE
	C	-	Assign^blockbuflen		A	File^blockbuflen
	C	-	filecode		A	filecode
	C		filename		A	* filename^addr
	C		logicalfilename		A	* logicalfilename^addr
	C	-	openaccess		A	openaccess
	C	-	openexclusion		A	openexclusion
	C	-	priext		A	priext
	C	+	recordlen		A	recordlen
	C	-	secext		A	secext
	C		Init^filefcb [1]			
	o	+	Set^abort^xfererr		o	File^abort^xfererr
C	A		breakhit	C	A	breakhit
A	A		checksum	A	A	checksum
	o	+	countxferred		o	countxferred
	o	+	crlf^break		o	crlf^break
	o	+	dupfile		o	* dupfile
	o	-	error		o	error
C	A		errorfile	C	A	* errorfile
	o	-	logioout		o	logioout
R	o		openerspid	C	A	* openerspid^addr
	o	-	physioout		o	physioout
	o	+	print^err^msg		o	print^err^msg
	o	+	prompt		o	prompt
	o	+	read^trim		o	read^trim
R	o		rcveof	R	o	rcveof
R	o		rcvopencnt	R	o	rcvopencnt
R	o		rcvuseropenreply	R	o	rcvuseropenreply
R	o		systemmessages	R	o	systemmessages
R	o		systemmessagemany	R	o	* systemmessagemany
C	A		traceback	C	A	traceback
	A		userflag		A	userflag
	o	+	write^fold		o	write^fold
	o	+	write^pad		o	write^pad
	o	+	write^trim		o	write^trim
		-			A	File^assignmask1 [2]
		-			A	assignmask2
		-		A	o	* bwdlinkfcb
		+			o	created
					A	* error^addr
		+			o	fileinfo
		-			o	fnum
					A	* fnum^addr
		-		A	o	* fwdlinkfcb
					A	* seqnum^addr
					A	* userflag^addr

[1] Init^filefcb should not be used for initializing the common^fcb.
[2] All assigns except filename are lost after a close^file.

SECTION 10

FORMATTER

The GUARDIAN operating system formatter provides the capability to format data on output and to convert data on input with a minimum of programming effort. The formatter consists of two procedures, which are called from user programs.

The formatter procedures are:

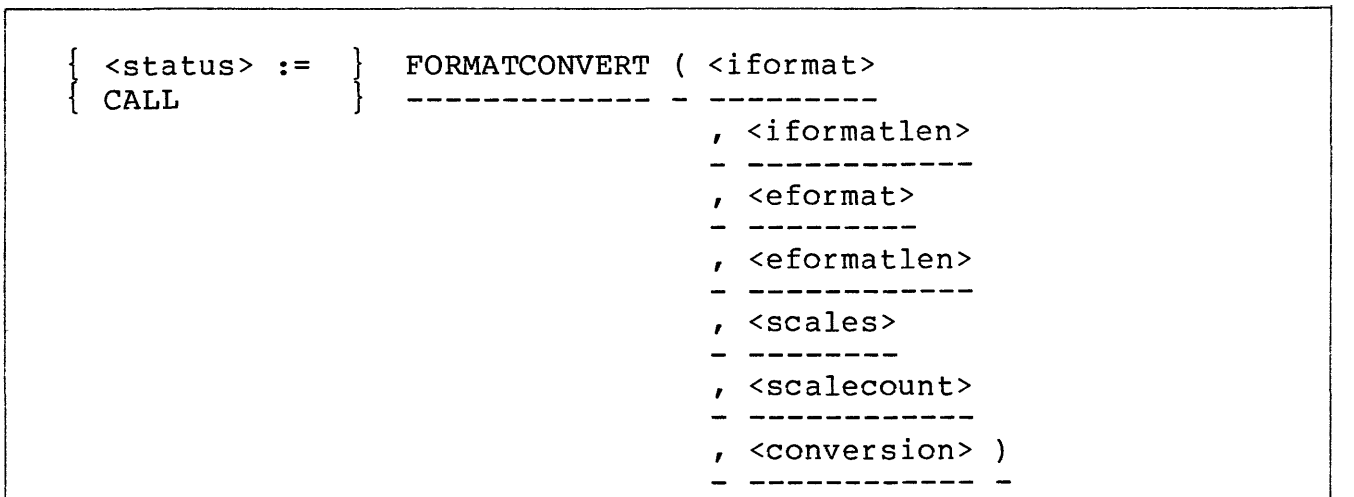
FORMATCONVERT	converts an external format to internal form for presentation to the FORMATDATA procedure.
FORMATDATA	performs conversion between internal and external representation of data as specified by a format, or performs conversion of data using the list-directed rules.

The decimal arithmetic package is required to use the formatter.

The floating-point arithmetic package is needed when using the "D", "E", and "G" edit descriptors for output or when floating-point variables are used.

The FORMATCONVERT procedure converts an external format to internal form for presentation to the FORMATDATA procedure.

The call to the FORMATCONVERT procedure is:



where

<status>, INT,

is a value indicating the outcome of FORMATCONVERT:

If > 0, indicates successful conversion. The value is the number of bytes in the converted format (<iformat>).

If = 0, indicates <iformatlen> was insufficient to hold the entire converted format.

If < 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>, STRING:ref,

is an array in which the converted format is to be stored. The contents of this array must be passed to the FORMATDATA procedure as an integer parameter, but FORMATCONVERT requires it to be 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 to be passed to FORMATDATA. (The area passed to FORMATDATA need not be in byte-addressable storage.)

→

<iformatlen>, INT,

is the length, in bytes, of the <iformat> array. If the converted format is longer than <iformatlen>, the conversion is terminated and a <status> value <= 0 is returned.

<eformat>, STRING:ref,

is the format string in external (ASCII) form.

<eformatlen>, INT,

is the length, in bytes, of the <eformat> string.

<scales>, INT:ref,

is an integer array. FORMATCONVERT processes the format from left to right, placing the scale factor (the number of digits that are to appear to the right of the decimal point) specified or implied by each repeatable edit descriptor into the next available element of <scales>, until the last repeatable edit descriptor has been converted or the maximum specified by <scalecount> is reached, whichever occurs first.

<scalecount>, INT:ref,

On call, the number of occurrences of the <scales> array.

On return, <scalecount> 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 <scalecount> maximum is reached, but continues to process the remaining edit descriptors and continues incrementing <scalecount>.

<conversion>, INT,

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 compact conversion, ignoring modifiers and decorations. The resulting format requires little storage space, but the execution time is twice as long as version 2 (below).

2 = Produce expanded form with modifiers and decorations. This requires additional storage space, but the execution time is half that of version 1 (above). The size required is approximately 10 times <eformatlen>.

Note: The <scales> parameter information was included to provide information needed by the ENFORM product. It is not of interest to most users of FORMATCONVERT. A variable initialized to zero should be supplied for <scales> and <scalecount> if this information is not of interest.

The FORMATDATA procedure performs conversion between internal and external representations of data, as specified by a format or the list-directed conversion rules.

The call to the FORMATDATA procedure is:

```
{ <error> := } FORMATDATA ( <buffer>
{ CALL       } ---------- - --------
                           , <bufferlen>
                           - -----------
                           , <bufferoccurs>
                           - --------------
                           , <length>
                           - --------
                           , <iformat>
                           - ---------
                           , <variablelist>
                           - --------------
                           , <variablelistlength>
                           - --------------------
                           , <flags> )
                           - ------- -
```

where

<error>, INT,

indicates the outcome of the call.

0 = Successful operation

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>, STRING:ref,

is a buffer or a series of contiguous buffers where the formatted output data is to be placed, or where the input data is found. The length, in bytes, of <buffer> must be at least <bufferlen> times <bufferoccurs>.

→

<bufferlen>, INT,

is the length, in bytes, of each buffer in the <buffer> array.

<bufferoccurs>, INT,

is the number of buffers in <buffer>.

<length>, 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 into the <length> parameter during input operation.

<iformat>, INT:ref,

is an integer array containing the internal format (as constructed by FORMATCONVERT).

<variablelist>, INT:ref,

is a 4- or 5-word entry for each array or variable. It 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; 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 to be 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 of the variable or array element in bytes, 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, is the byte address of the null value.
                 If = 0, no null value for this variable.


<variablelistlength>, INT,

   is the number of <variablelist> entries passed in this
   call.

<flags>, INT:value,

   <15:15> = Input:

             If 0, FORMATDATA performs output operations.
             If 1, FORMATDATA performs input operations.

   <04:04> = Null value passed:

             If 0, each <variablelist> item is a 4-word
                   group.
             If 1, each <variablelist> item is a 5-word
                   group.

   <03:03> = P-Relative (<iformat> array):

             If 0, the <iformat> array is G-relative.
             If 1, the <iformat> array is P-relative.

   <02:02> = List-directed (refer to "List-Directed
             Formatting" at the end of this section):

             If 0, apply the format-directed operation.
             If 1, apply the list-directed operation.
```

Errors

267 BUFFER OVERFLOW
FORMATDATA required access to a character before start of buffer or outside buffer to interpret an edit descriptor.

268 NO BUFFER
FORMATDATA required new buffer, but current buffer was the last one supplied. Correct format or increase buffer space.

270 FORMAT LOOPBACK
When FORMATDATA reached end of a format which contained no repeatable edit descriptors, data items remained to be processed. This error would cause an infinite loop if not detected. Include repeatable edit descriptors in the format, or reduce number of data items.

271 EDIT ITEM MISMATCH
In format-directed operation, edit descriptor was matched to data element of incompatible type. (For example, "G" edit descriptor was associated with string data element on output, or any edit descriptor except "A" was associated with string data element on input.) In list-directed input, numeric data element was repeated using r*c form, and some data element after first element to which this form applied was a string-type element. Correct format, or correct data list to include missing items or delete extra ones.

272 ILLEGAL INPUT CHARACTER
Numeric input field contained character that was inappropriate for corresponding edit descriptor. For example, non-numeric character was entered in field being interpreted according to "I" edit descriptor. Correct format or data list.

273 BAD FORMAT
Format containing edit descriptor that is valid for output, but not for input, was used for input. For example, I5.5 is invalid for input. Correct the format.

274 NUMERIC OVERFLOW
Numeric value was too small or too large to place in corresponding data element. Change format or correct numerical calculations.

EXAMPLE

This example shows how to use the Formatter procedures for some simple output editing. It illustrates the setup for the variable list and the use of the <length> values returned from FORMATDATA.

```
PROC EXAMPLE MAIN;
   BEGIN
!
!  THIS STRUCTURE DEFINES THE 4-WORD FORM OF VARIABLE LIST ENTRY
!  (THE ONE WITHOUT THE NULL VALUE POINTER FIELD).
!
   STRUCT VLE^REF (*);
      BEGIN
      INT    ELE^PTR ;
      STRING ELE^SCALE, ELE^TYPE ;
      INT    ELE^LEN, ELE^OCCURS ;
      END ;
!
!  THIS DEFINE PROVIDES ONE WAY TO INITIALIZE THE FIELDS OF A
!  VARIABLE LIST ENTRY.  THE SCALE, TYPE, LENGTH, AND OCCURS VALUES
!  MUST BE CONSTANTS TO BE ABLE TO USE IT.
!
   DEFINE VLE^INIT (ENT, V, SCALE, TYPE, LEN, OCCURS) =
              BEGIN
              ENT ':=' [ 0, SCALE '<<' 8 '+' TYPE, LEN, OCCURS ] ;
              ENT.ELE^PTR := @V ;
              END  #;
!
!  THIS STRUCTURE DEFINES A BUFFER TO MAKE IT EASIER TO CREATE
!  AN ARRAY OF BUFFERS.
!
   LITERAL BUF^LEN = 100 ;
   STRUCT BUF^REF (*) ;
      BEGIN
      STRING BYTES [0:BUF^LEN-1] ;
      END ;
!
!  THE EXAMPLE FORMAT IN EXTERNAL (ASCII) FORM.
!
   LITERAL EFORMATLEN = 60 ;
   STRING .EFORMAT [0:EFORMATLEN] :=
             "20X,'SAMPLE OUTPUT' // I5,2X,F10.3,5(2X,I2),5X,A" ;
!
!  STORAGE FOR THE INTERNAL FORM OF THE FORMAT.
!
   LITERAL IFORMATLEN = 200 ;
   INT    .WFORMAT [0:IFORMATLEN/2] ;
   STRING .IFORMAT := @WFORMAT '<<' 1 ;
```

```
!
!  ARRAY OF BUFFERS AND OF THE LENGTH USED IN EACH.
!
   LITERAL NUM^BUFS = 5 ;
   STRUCT .BUFFERS (BUF^REF) [0:NUM^BUFS-1] ;
   INT    .BUF^LENS [0:NUM^BUFS-1] ;
!
!  VARIABLE LIST ARRAY.
!
   STRUCT .VLIST (VLE^REF) [0:3] ;
!
!  DATA FOR THE EXAMPLE.
!
   INT      INT^16 :=  7 ;
   FIXED(2) QUAD := -437.57F ;
   INT(32) .INT^32^ARRAY [0:4] := [ 1D, 1D, 2D, 3D, 5D ] ;
   STRING  .CHARS [0:10] := "DEMO STRING" ;
!
!  MISCELLANEOUS DATA.
!
   INT     .FILENAME [0:11] ;
   INT      FILENO ;
   INT      SCALES, ERROR, I ;
!
!  INITIALIZATION
!
   CALL MYTERM (FILENAME) ;
   CALL OPEN (FILENAME, FILENO) ;
!
!  CONVERT THE FORMAT TO INTERNAL FORM.
!  NOTE THE WAY TO IGNORE THE SCALE INFORMATION.
!
   SCALES := 0 ;
   ERROR := FORMATCONVERT (IFORMAT, IFORMATLEN, EFORMAT, EFORMATLEN,
                           SCALES, SCALES, 1) ;
   IF ERROR <= 0 THEN BEGIN
                        !  HERE IF ERROR IN FORMAT
                      END ;
!
!  SET UP THE VARIABLE LIST ENTRIES, BOTH BY USING THE DEFINE
!  AND BY SEPARATE STORES INTO THE ITEM FIELDS.
!
   VLE^INIT (VLIST[0], INT^16, 0, 2, 2, 1) ;
                ! SCALE 0, TYPE 2, LEN 2 BYTES, 1 OCCURRENCE
   VLE^INIT (VLIST[1], QUAD, 2, 6, 8, 1) ;
                ! SCALE 2, TYPE 6, LEN 8 BYTES, 1 OCCURRENCE
```

```
   VLIST[2].ELE^PTR    := @INT^32^ARRAY ;       ! VARIABLE ADDRESS
   VLIST[2].ELE^SCALE  := 0 ;                   ! SCALE 0
   VLIST[2].ELE^TYPE   := 4 ;                   ! TYPE 4
   VLIST[2].ELE^LEN    := 4 ;                   ! LENGTH 4 BYTES
   VLIST[2].ELE^OCCURS := 5 ;                   ! 5 OCCURRENCES

   VLE^INIT (VLIST[3], CHARS, 0, 0, 11, 1) ;
               ! SCALE 0, TYPE 0, LEN 11 BYTES, 1 OCCURRENCE
!
!  EDIT THE DATA INTO THE BUFFERS.
!
   ERROR := FORMATDATA (BUFFERS, BUF^LEN, NUM^BUFS, BUF^LENS, WFORMAT,
                        VLIST, 4, 0) ;
   IF ERROR <> 0 THEN BEGIN
                        !  HERE IF ERROR IN DATA CONVERSION
                      END ;
!
!  WRITE THE BUFFERS USED TO THE TERMINAL.
!
   I := 0 ;
   WHILE  I <= NUM^BUFS  AND  BUF^LENS[I] >= 0  DO  BEGIN
             CALL WRITE (FILENO, BUFFERS[I], BUF^LENS[I]) ;
             I := I + 1 ;
             END ;
!
!  THE OUTPUT PRODUCED IS THE THREE LINES SHOWN BELOW.
!  THE | CHARACTER IS USED TO SHOW THE BUFFER LIMITS INDICATED
!  IN THE BUF^LENS ARRAY:
!
!  |                     SAMPLE OUTPUT|
!  ||
!  |    7    -437.570   1   1   2   3   5     DEMO STRING|
!
   CALL STOP ;

   END ;   !  EXAMPLE
```

FORMAT-DIRECTED FORMATTING

The principal parameters to the Formatter are a list of "data elements", an array of buffers, and a "format".

The format is a list of "edit descriptors", separated by commas, which are translated into internal form by FORMATCONVERT for presentation to FORMATDATA. Edit descriptors may optionally be preceded by one or more "modifiers" and/or "decorations", enclosed in brackets ([]), that specify additional field formatting. The FORMATCONVERT procedure converts the external data into an internal form for presentation to the FORMATDATA procedure.

The FORMATDATA procedure matches each data element with its associated edit descriptor, which specifies how it is to be displayed for output or how the buffer contents are to be interpreted for input. FORMATDATA proceeds through the list, from left to right, of edit descriptors in the order in which they were presented. If an edit descriptor is a "non-repeatable" item, FORMATDATA processes it directly; if an edit descriptor is a "repeatable" item, FORMATDATA obtains the next data element from the data list and performs the data conversion specified by the edit descriptor. This processing continues until the data list is exhausted.

Exceptions to the left-to-right processing are the repeat factor and format loopback. Any edit descriptor, or groups of edit descriptors enclosed in parentheses, can be applied repeatedly to a number of data values by a positive-integer "repeat factor" preceding the descriptor or group. If the end of the format is reached with unprocessed data elements remaining, "format loopback" selects a portion of the format which is to be interpreted again.

The <variablelist> defines a sequence of variables or arrays which are to be processed by the FORMATDATA procedure. Each variable or element of an array in the <variablelist> is referred to as a "data element".

Format Characteristics

A "format" directs the operation of the FORMATDATA procedure's editing between the internal representation and external representation of data.

The form of a format is:

Edit descriptors are of two types: those that specify the conversion of data values (repeatable) and those that do not (non-repeatable). The effect of repeatable edit descriptors can be altered through the use of modifiers or decorations, which are enclosed in brackets ([]) preceding the edit descriptors to which they refer. Within a format, all edit descriptors except buffer control descriptors must be separated by commas. Buffer control descriptors have the dual function of edit descriptors and format separators, and need not be set off by commas.

```
                { fmt-item    }   [ separator   fmt-item    ]
format:         {             }   [                         ] . . .
                { b-separator }   [ [separator] b-separator ]

                { non-repeatable-edit-descriptor }
fmt-item:       {                                }
                {         field-group            }

field-group:  [repeat]  [mods]  { group-spec }

                { repeatable-edit-descriptor }
group-spec:     {                            }
                {     "(" format  ")"        }

repeat:       an unsigned, non-zero integer

                     { modifier  }
mods:         "["    {           } , . . .  "]"
                     { decoration }
```

→

```
separator:    { , | / | : }

b-separator:  { / | : }


non-repeatable          repeatable
edit descriptors     edit descriptors    modifiers    decorations

  BN       SP          A      G         BN   OC     { P }
  BZ       SS          D      I         BZ   RJ     { M }      { A }
  H        T           E      L         FL   SS     { Z } ...  { F }
  string   TL          F      M         LJ          { N }      { P }
  P        TR                                      { O }
  S        X


Some sample formats:

    I5,F10.2

    "        NAME         EXTENSION",//,(A20,3X,I4)

    3("  ITEM     ACTIVITY",10X),//,3(M<99-9999>,5X,[BZ]I6,1X,10X)
```

FORMATDATA matches each data element with its associated edit descriptor, which specifies how it is to be displayed for output or how the buffer contents are to be interpreted for input. FORMATDATA proceeds through the list (from left to right) of edit descriptors in the order in which they were presented:

1. If an edit descriptor is a repeatable item, FORMATDATA obtains the next data element from the data list and performs the data conversion specified by the edit descriptor.

2. If an edit descriptor is a non-repeatable item, FORMATDATA processes it directly.

This processing continues until the data list is exhausted. If there are any data list items, there must be at least one repeatable edit descriptor in the format.

The interpretation of the format terminates if any of these conditions are met:

1. FORMATDATA encounters a repeatable edit descriptor in the format and there are no remaining data elements.

2. FORMATDATA reaches the end of the format and there are no remaining data elements.

3. FORMATDATA encounters a colon edit descriptor in the format and there are no remaining data elements.

A format is interpreted from left to right with the following exceptions:

1. If a field group contains a repeat factor, then the group specifications are processed the number of times indicated by the repeat factor before continuing with the following specifications.

2. If FORMATDATA reaches the end of the format and data elements remain, format loopback occurs. Format loopback performs the following steps:

    a. The current buffer is terminated.

    b. A new buffer is obtained.

    c. The format is examined backwards (from right to left). If a right parenthesis which is not part of a string or Hollerith descriptor is encountered, the matching left parenthesis is found, and the format interpretation resumes at the left parenthesis. If a repeat factor precedes this left parenthesis, processing resumes at the repeat factor. If the beginning of the format is reached and no right parenthesis is found, the format interpretation resumes at the beginning.

    d. Reverse examination of the format position has no effect on the scale factor (set by P), the sign control (set by S,SP, or SS), or the blank control (set by BN or BZ). The condition in effect at the end of the format continues until altered by one of the controlling edit descriptors.

EDIT DESCRIPTORS

Edit descriptors are of two types: those that specify the conversion of data values (repeatable), and those that do not (non-repeatable). The effect of repeatable edit descriptors can be altered through the use of modifiers or decorations, which are enclosed in brackets ([]) preceding the edit descriptors to which they refer. Within a format, all edit descriptors except buffer control descriptors must be separated by commas. Buffer control descriptors have the dual function of edit descriptors and format separators, and need not be set off by commas.

Summary of Non-Repeatable Edit Descriptors

The edit descriptors that are not associated with data items are of six subtypes:

- Tabulation

  1. Tn - Tab absolute to "nth" character position
  2. TRn - Tab right
  3. TLn - Tab left
  4. nX - Tab right (same as TR)

- Literals

  1. Alphanumeric string enclosed in apostrophes (') or quotation marks (")
  2. Hollerith descriptor (nH followed by "n" characters)

- Scale factor specification

  1. P - Implied decimal point in a number

- Optional plus control

  These descriptors provide control of the appearance of an optional plus sign for output formatting. They have no effect on input.

  1. S - Do not supply a plus
  2. SP - Supply a plus
  3. SS - Do not supply a plus

- Blank interpretation control

  1. BN - Blanks ignored (unless entire field is blank)
  2. BZ - Blanks treated as zeros

- Buffer control

  1. / - Terminate the current buffer, and then obtain a new one
  2. : - Terminate formatting if no data elements remain

Summary of Repeatable Edit Descriptors

Repeatable edit descriptors direct the Formatter to obtain the next data list element and perform a conversion between internal and external representation. They may be preceded by modifiers or decorations which may alter the interpretation of the basic edit descriptor. Modifiers and decorations apply only to output conversion. They are allowed but ignored for input.

The repeatable edit descriptors are:

1. A - Alphanumeric (ASCII)
2. D,E - Exponential form
3. F - Fixed form
4. G - General (E or F format depending on magnitude of data)
5. I - Integer
6. L - Logical
7. M - Mask formatting

Summary of Modifiers

Modifiers are codes that are used to alter the results of the formatting prescribed by the edit descriptors to which they are attached. They are:

1. BN, BZ - Field blanking (if null, or zero)
2. FL - Fill character specification
3. LJ, RJ - Left and right justification
4. OC - Overflow character modifier
5. SS - Symbol substitution

Note: Table 10-1 in the "Modifiers" section displays which modifiers may be used in combination with what edit descriptor.

Summary of Decorations

Decorations specify alphanumeric strings that can be added to a field either before basic formatting is begun or after it is finished. A decoration consists of one or more codes that specify the condition(s) under which the string is to be added (based on the value of the data element or the occurrence overflow of the external field):

1. M - Minus
2. N - Null
3. O - Overflow
4. P - Plus
5. Z - Zero

followed by a code that describes the position of the special editing:

1. A (absolute) - at a specific character position within the field
2. F (floating) - at the position the basic formatting finished
3. P (prior) - at the position the basic formatting would have started

followed by the character string that is to be included in the field if the stated conditions are met.

NON-REPEATABLE EDIT DESCRIPTORS

The following descriptions show the form, function, and requirements for each of the non-repeatable edit descriptors.

Tabulation Descriptors

The tabulation descriptors specify the position at which the next character is transmitted to or from the buffer. This allows portions of a buffer to be processed in an order other than strictly left to right, and permits processing of the same portion of a buffer more than once.

The forms of the tabulation descriptors are:

Tn	TLn	TRn	nX
where "n" is an unsigned integer constant.			

Each of these edit descriptors alters the current position but has no other effect.

"Tn" indicates that the transmission of the next character to or from a buffer is to occur at the "n"th character position. The first character of the buffer is numbered 1.

"TLn" indicates that the transmission of the next character to or from the buffer is to occur "n" positions to the left of the current position.

"TRn" indicates that the transmission of the next character to or from the buffer is to occur "n" positions to the right of the current position.

"nX" indicates that the transmission of the next character to or from the buffer is to occur "n" positions to the right of the current position.

The current position may be moved beyond the limits of the current buffer (that is, become less than or equal to zero, or greater than <bufferlen>) without an error resulting, provided that no attempt is made by a subsequent edit descriptor to transmit data to or from a position outside the current buffer.

Tab descriptors may not be used to advance to later buffers nor to return to previous ones.

Examples:

Data List Values

```
 100
-1000.49F
 "HELLO"
```

	Format	Results
No tabs	I3,E12.4,A5	100 0.1000E+04HELLO
X	I3,E12.4,1X,A5	100 0.1000E+04 HELLO
TL	I3,E12.4,TL3,A5	100 0.1000EHELLO
TR	I3,E12.4,TR5,A5	100 0.1000E+04 HELLO
T	I3,E12.4,T3,A5	10HELLO1000E+04

Note: The "/\" marker is used to denote the boundaries of the output field.

Literal Descriptors

Literal descriptors are alphanumeric strings in either of the forms:

```
dc c c  ... c d     OR     nHc c c  ... c
  1 2 3      n                1 2 3      n
```

where

d = either an apostrophe (') or a quotation mark ("); the same character must be used for both the opening and closing delimiters.

c = any ASCII character.

n = an unsigned nonzero integer constant specifying the number of characters in the string; "n" may not exceed 255.

On input, a literal descriptor is treated as "nX".

A literal edit descriptor causes the specified character string to be inserted in the current buffer beginning at the current position. It advances the current position "n" characters.

In a quoted literal form, if the character string to be represented contains the same character that is used as the delimiter, two consecutive characters are used to distinguish the data character from the delimiter.

For example,

To represent:	can't	Use:	'can''t'	or	"can't"
	"can't"		'"can''t"'	or	"""can't"""

In the Hollerith constant form, the number of characters in the string (including blanks) must be exactly equal to the number preceding the letter "H". There are no delimiter characters, so the characters are supplied exactly as they should appear in the buffer.

For example,

To represent:	can't	Use:	5Hcan't

Scale Factor Descriptor (P)

The form of a scale factor descriptor is:

```
      nP

  n = an optionally signed integer in the range of -128 to +127.
```

The value of the scale factor is zero at the beginning of execution of the FORMATDATA procedure. Any scale factor specification remains in effect until a subsequent scale specification is processed. The scale factor applies to the "D", "E", "F", and "G" edit descriptors, affecting them in the following manner:

1. On input, with "D", "E", "F", and "G" edit descriptors (provided no exponent exists in the external field), the scale factor effect is that the externally represented number equals the internally represented number multiplied by 10**n.

2. On input, with "D", "E", "F", and "G" edit descriptors, the scale factor has no effect if there is an exponent in the external field.

3. On output, with "D" and "E" edit descriptors, the mantissa of the quantity to be produced is multiplied by 10**n and the exponent is reduced by n.

4. On output, with the "F" edit descriptor, the scale factor effect is that the externally represented number equals the internally represented number multiplied by 10**n.

5. On output, with the "G" edit descriptor, the effect of the scale factor is suspended unless the magnitude of the datum to be processed is outside the range that permits the use of an "F" edit descriptor. If the use of the "E" edit descriptor is required, the scale factor has the same effect as with the "E" output processing.

Optional Plus Descriptors (S, SP, SS)

Optional plus descriptors may be used to control whether optional plus characters appear in numeric output fields. In the absence of explicit control, the formatter does not produce any optional plus characters.

The forms of the optional plus descriptors are:

S	SP	SS

These descriptors have no effect upon input.

If the "S" descriptor is encountered in the format, the formatter does not produce a plus in any subsequent position that normally contains an optional plus.

If the "SP" descriptor is encountered in the format, the formatter produces a plus in any subsequent position that normally contains an optional plus.

The "SS" descriptor is the same as "S" (above).

An optional plus is any plus except those appearing in an exponent.

Blank Descriptors (BN, BZ)

The blank descriptors have the following form:

BN	BZ

These descriptors have no effect on output.

The "BN" and "BZ" descriptors may be used to specify the interpretation of blanks, other than leading blanks, in numeric input fields. At the beginning of execution of the FORMATDATA procedure, non-leading blank characters are ignored.

If a "BZ" descriptor is encountered in a format, all non-leading blank characters in succeeding numeric input fields are treated as zeros.

If a "BN" descriptor is encountered in a format, all blank characters in succeeding numeric input fields are ignored. The effect of ignoring blanks is to treat the input field as if all blanks had been removed, the remaining portion of the field right-justified, and the blanks reinserted as leading blanks. However, a field of all blanks has the value zero.

The "BN" and "BZ" descriptors affect the "D", "E", "F", "G", and "I" edit descriptors only.

Buffer Control Descriptors (/, :)

There are two edit descriptors used for buffer control:

/ = indicates the end of data list item transfer on the current buffer and obtains the next buffer. The current position is moved to 1 in preparation for processing the next buffer. : = indicates termination of the formatting provided there are no remaining data elements.

- To clarify, the operation of the slash (/) is as follows for any positive integer n:

  1. If n consecutive slashes appear at the end of a format, this causes n buffers to be skipped.

  2. If n consecutive slashes appear within the format, this causes n-1 buffers to be skipped.

- The colon (:) is used to conditionally terminate the formatting. If there are additional data list items, the colon has no effect. The colon can be of use when data items are preceded by labels, as in the following example:

```
10(' NUMBER ',I1,:/)
```

This group of edit descriptors is preceded by a repeat factor that specifies the formatting of ten data items, each one to be preceded by the label NUMBER. If there are fewer than ten data items in the data list, formatting terminates immediately after the last value is processed. If the colon were not present, formatting would continue until the "I" edit descriptor was encountered for the fourth time. This means the fourth label would be added before the formatting was terminated.

For example:

Data Items:

```
1
2
3
```

Format:

With colon	Without colon
10('NUMBER ',I1,:/)	10('NUMBER ',I1,/)

Results:

|NUMBER 1|	|NUMBER 1|
|NUMBER 2|	|NUMBER 2|
|NUMBER 3|	|NUMBER 3|
	|NUMBER |

Note: The '|' character is used to denote the boundaries of the output field.

REPEATABLE EDIT DESCRIPTORS

The following descriptions give the form, function, and requirements for each of the edit descriptors that specify formatting of data fields. The following edit descriptors may be preceded by an unsigned integer repeat factor to specify identical formatting for a number of values in the data list.

The following descriptions of the operation of repeatable edit descriptors apply when no decorations or modifiers are present.

"A" Edit Descriptor

This edit descriptor is used to move characters between the buffer and the data element without conversion. This is normally used with ASCII data.

The "A" edit descriptor has one of the following forms:

```
      Aw          OR          A

where

   w = an unsigned integer constant that specifies the width,
       in characters, of the field and may not exceed 255.
       The field processed is the next "w" characters starting
       at the current position.

       If "w" is not present, the field width is equal to
       the actual number of bytes in the associated data element.

       After the field is processed, the current position is
       advanced by "w" characters.
```

On output, the operation of the "A" edit descriptor is as follows:

1. The number of characters specified by "w", or the number of characters in the data element, whichever is less, is moved to the external field. The transfer starts at the left character of both the data element and the external field unless an "RJ" modifier is affecting the descriptor, in which case the transferring of characters begins with the right character of each.

2. If "w" is less than the number of characters in the data element, the field overflow condition is set.

3. If "w" is greater than the number of characters in the data element, the remaining characters in the external field are filled with spaces (unless another fill character is specified by the "FL" modifier).

It is not mandatory that the data element be of type character. For example, an INTEGER(16) element containing the octal value %015536 corresponds to the ASCII characters "ESC" and "^", which can be output to an ADM-2 terminal using an A2 descriptor to control a blinking field on the screen.

Examples:

Format	Data Value	External Field	
A	'WORD'	|WORD|	
A4	'WORD'	|WORD|	
A3	'WORD'	|WOR|	(overflow set)
[RJ]A3	'WORD'	|ORD|	(overflow set)
A5	'WORD'	|WORD |	
[RJ]A5	'WORD'	| WORD|	
A	%044111	|HI|	

Notes: In the last example, the data value was stored in a 2-byte INTEGER.

The "|" character is used to denote the boundaries of the output field.

On input, the operation of the "Aw" edit descriptor is as follows:

1. The number of characters specified by "w", or the number of characters contained in the data element, whichever is less, is moved from the external field to the data element. The transfer begins at the left character of both the data element and the external field.

2. If "w" is less than the number of characters in the data element, the remaining characters in the field are truncated.

3. If "w" is greater than the number of characters contained in the data element, the remaining characters of the data element are filled with spaces.

Examples:

External Field	Format	Data Item Length	Data Element Value
|HELLO|	A5	5 characters	'HELLO'
|HELLO|	A3	3 characters	'HEL'
|HELLO |	A6	6 characters	'HELLO '
|HELLO|	A5	6 characters	'HELLO '

Note: The "|" character is used to denote the boundaries of the input field.

"D" Edit Descriptor

The exponential edit descriptor is used to display or interpret data in floating-point form, usually used when data values have extremely large or extremely small magnitude.

The "D" edit descriptor is of the form:

```
        Dw.d

   This descriptor is identical to the "Ew.d" descriptor.
```

Note: This edit descriptor is used in the same manner as the "E" edit descriptor (below).

To use the "D" edit descriptor for output, floating-point firmware is required.

"E" Edit Descriptor

The exponential edit descriptor is used to display or interpret data in floating-point form. It is usually used when data values have extremely large or extremely small magnitude.

The "E" edit descriptor has one of the following forms:

Ew.d OR Ew.dEe

where

w = an unsigned integer constant that defines the total field width (including the exponent) and may not exceed 255. The field processed is the "w" characters starting at the current position. After the field is processed, the current position is advanced by "w" characters.

d = an unsigned integer constant that defines the number of digits that are to appear to the right of the decimal point in the external field.

e = an unsigned integer constant that defines the number of digits in the exponent. If "Ew.d" is used, "e" takes the value 2.

The input field consists of an optional sign, followed by a string of digits optionally containing a decimal point. A decimal point appearing in the input field overrides the portion of the descriptor that specifies the decimal point location. However,if the decimal point is omitted, the rightmost "d" digits of the string, with leading zeroes assumed if necessary, are interpreted as the fractional part of the value represented. The string of digits may be of any length. Those beyond the limit of precision of the internal representation are ignored. The basic form may be followed by an exponent in one of the following forms:

1. Signed integer constant.

2. "E" followed by zero or more blanks, followed by an optionally signed integer constant.

3. "D" followed by zero or more blanks, followed by an optionally signed integer constant.

An exponent containing a "D" is processed identically to an exponent containing an "E".

On output, the field (for a scale factor of zero) appears in the following form:

$$\left\{\begin{matrix}[+]\\[-]\end{matrix}\right\}\ [0].n_1 n_2 \ \ldots\ n_d \quad E \left\{\begin{matrix}+\\-\end{matrix}\right\}\ e_1 e_2 \ldots e_e$$

where

$\left\{\begin{matrix}[+]\\[-]\end{matrix}\right\}$ indicates an optional plus or a minus.

$n_1 n_2 \ldots n_d$ are the "d" most significant digits of the value of the datum after rounding.

E signals the start of the decimal exponent.

$\left\{\begin{matrix}+\\-\end{matrix}\right\}$ indicates that a plus or minus is required.

$e_1 e_2 \ldots e_e$ are the "e" most significant digits of the exponent.

The sign in the exponent is always displayed. If the exponent is zero, a plus sign is used.

If the datum is negative, the minus sign is always displayed. If the datum is positive (or zero), the display of the plus sign is dependent on the last optional plus descriptor processed.

The zero preceding the decimal point is normally displayed, but may be omitted to prevent field overflow.

Decimal normalization is controlled by the scale factor established by the most recently interpreted "nP" edit descriptor. If $-d < n \leq 0$, the output value has $|n|$ leading zeroes, and $(d-|n|)$ significant digits follow the decimal point; if $0 < n < d+2$, the output value has n significant digits to the left of the decimal point and d-n+1 digits to the right. If the number of characters produced exceeds the field width or if an exponent exceeds its specified length using the "Ew.dEe" field descriptor, the entire field of width "w" is filled with asterisks. However, if the field width is not exceeded when optional characters are omitted, the field is displayed without the optional characters.

Because all characters in the output field are included in the field width, "w" must be large enough to accommodate the exponent, the decimal point, and all digits and the algebraic sign of the base number.

Examples of output:

Format	Data Value	Result
E12.3	8.76543 x 10	| 0.877E-05|
E12.3	-0.55555	| -0.556E+00|
E12.3	123.4567	| 0.123E+03|
E12.6E1	3.14159	| 0.314159E+1|

Note: The "|" character is used to denote the boundaries of the output field.

To use the "E" edit descriptor for output, floating-point firmware is required.

Examples of input:

External Field	Format	Data Element Value
| 0.100E+03|	E12.3	100
| 100.05 |	E12.5	100.05
| 12345|	E12.3	12.345

Note: The "|" character is used to denote the boundaries of the output field.

"F" Edit Descriptor

The fixed-format edit descriptor is used to display or interpret data in fixed point form.

The "F" edit descriptor has the following forms:

Fw.d OR Fw.d.m

where

w = an unsigned integer constant that defines the total field width and may not exceed 255. The field processed is the "w" characters starting at the current position. After the field is processed, the current position is advanced by "w" characters.

d = an unsigned integer constant that defines the number of digits that are to appear to the right of the decimal point in the external field.

→

m = an unsigned integer constant that defines the number of digits that must be present to the left of the decimal point on input.

On input, the "Fw.d" edit descriptor is the same as the "Ew.d" edit descriptor.

The output field consists of blanks if necessary, followed by a minus if the internal value is negative or an optional plus otherwise, followed by a string of digits that contains a decimal point and represents the magnitude of the internal value, as modified by the established scale factor and rounded to the "d" fractional digits. If the magnitude of the value in the output field is less than one, there are no leading zeros except for an optional zero immediately to the left of the decimal point. The optional zero must appear if there would otherwise be no digits in the output field. If the "Fw.d.m" form is used, leading zeroes are supplied if needed to satisfy the requirement of m digits to the left of the decimal point.

Examples:

Format	Data Value	Result
F10.4	123.4567	123.4567|
F10.4	0.000123	0.0001|
F10.4.3	-4.56789	-004.5679|

Note: The "|" character is used to denote the boundaries of the output field.

"G" Edit Descriptor

The general format edit descriptor can be used in place of either the "E" or the "F" edit descriptor, since it has a combination of the capabilities of both.

The "G" edit descriptor has either of the forms:

```
   Gw.d          OR          Gw.dEe

where

   w = an unsigned integer constant that defines the total
       field width and may not exceed 255.  The field
       processed is the "w" characters starting at the current
       position.  After the field is processed, the current
       position is advanced by "w" characters.

   d = an unsigned integer constant that defines the number of
       significant digits that are to appear in the external
       field.

   e = an unsigned integer constant that defines the number of
       digits in the exponent, if one is present.
```

On input, the "G" edit descriptor is the same as the "E" edit descriptor.

The method of representation in the output field depends on the magnitude of the datum being processed, as follows:

Magnitude of Data		
Not Less Than	Less Than	Equivalent Conversion Effected
	0.1	Ew.d or Ew.dEe
0.1	1.0	F(w-n).d,n(' ')
1.0	10.0	F(w-n).(d-1),n(' ')
10.0	100.0	F(w-n).(d-2),n(' ')
.	.	.
.	.	.
.	.	.
10 ** (d-2)	10 ** (d-1)	F(w-n).1,n(' ')
10 ** (d-1)	10 ** d	F(w-n).0,n(' ')
10 ** d		Ew.d or Ew.dEe

The value of n is 4 for "Gw.d" format and (e+2) for "Gw.dEe" format. The "n(' ')" used in the above example indicates nth number of blanks. If the "F" form is chosen, then the scale factor is ignored. The following comparison between "F" formatting and "G" formatting is given by way of illustration:

Value	"F" F13.6 Conversion	"G" G13.6 Conversion
.01234567	0.012346	0.123457E-01
.12345678	0.123457	0.123457
1.23456789	1.234568	1.23457
12.34567890	12.345679	12.3457
123.45678900	123.456789	123.457
1234.56789000	1234.567890	1234.57
12345.67890000	12345.678900	12345.7
123456.78900000	123456.789000	123457.
1234567.89000000	**********	0.123457E+07

When an overflow condition occurs in a numeric field, the field is filled with asterisks (in the absence of any specification to the contrary by an overflow decoration), as shown above.

Note: The "|" character is used to denote the boundaries of the output field.

To use the "G" edit descriptor for output, floating-point firmware is required.

"I" Edit Descriptor

The integer edit descriptor is used to display or interpret data values in an integer form.

The "I" edit descriptor has the following forms:

Iw OR Iw.m

where

w = an unsigned integer constant that defines the total width of the field and may not exceed 255. The field processed is the "w" characters starting at the current position. After the field is processed, the current position is advanced by "w" characters.

m = an unsigned integer constant that defines the number of digits that must be present in the field on output.

On output, the external field consists of zero or more leading blanks (followed by a minus if the value of the internal datum is negative, or an optional plus otherwise), followed by the magnitude of the internal value in the form of an unsigned integer constant without leading zeroes. An integer constant always consists of at least one digit. The output from an "Iw.m" edit descriptor is the same as the above, except that the unsigned integer constant consists of at least "m" digits and, if necessary, has leading zeroes. The value of "m" must not exceed the value of "w". If "m" is zero and the internal datum is zero, the output field consists only of blank characters, regardless of the sign control in effect.

Examples:

Format	Data Value	Result
I7	100	| 100|
I7.2	-1	| -01|
I7.6	100	| 000100|
I7.6	-1	|-000001|

Note: The "|" character is used to denote the boundaries of the output field.

On input, an "Iw.m" edit descriptor is treated identically to an "Iw" edit descriptor. The edit descriptors "Iw" and "Iw.m" indicate that the field to be edited occupies "w" positions. In the input field, the character string must be in the form of an optionally signed integer constant, except for the interpretation of blanks. Leading blanks on input are not significant, and the interpretation of any other blanks is determined by blank control descriptors ("BN" and "BZ").

Examples:

External Field	Format	Data Element Value
| 100|	I7	100
| -01|	I7	-1
| 1 |	I7	1
| 1 |	BZ,I7	1000
| 1 2 |	BZ,I7	10200
| 1 2 |	BN,I7	12

Note: The "|" character is used to denote the boundaries of the output field.

"L" Edit Descriptor

The logical edit descriptor is used to display or interpret data in logical form.

The "L" edit descriptor has the form:

```
    Lw

  where

     w = an unsigned integer constant that defines the width of
         the field and may not exceed 255.  The field processed
         is the "w" characters starting at the current position.
         After the field is processed, the current position is
         advanced by "w" characters.
```

On output, the "L" edit descriptor causes the associated data element to be evaluated in a logical context, and a single character is inserted right-justified in the output field. If the data value is null, the character is blank. If the data value is zero, the character is "F"; for all other cases, the character is "T".

Examples:

Format	Data Value	Result
L2	-1	| T|
L2	15769	| T|
L2	0	| F|

Note: The "|" character is used to denote the boundaries of the output field.

For input, the input field consists of optional blanks, optionally followed by a decimal point, followed by a "T" for true (logical value -1) or "F" for false (logical value 0). The "T" or "F" may be followed by additional characters in the field. The logical constants .TRUE. and .FALSE. are acceptable input forms.

Examples:

External Field	Format	Data Element Value
|T|	L7	-1
|F|	L7	0
|.TRUE.|	L7	-1
|.FALSE.|	L7	0
|TUGBOAT|	L7	-1
|FARLEY |	L7	0

Note: The "|" character is used to denote the boundaries of the output field.

"M" Edit Descriptor

The mask formatting edit descriptor edits either alphanumeric or numeric data according to an editing pattern or mask. Special characters within the mask indicate where digits in the data are to be displayed; other characters are duplicated in the output field as they appear in the mask.

The "M" edit descriptor has the form:

```
    M<mask>

  where

    <mask> = a character string enclosed in a pair of apostrophes
             (´), a pair of quotation marks ("), or less-than and
             greater-than symbols (<>). The string supplied must
             not exceed 255.
```

The "M" edit descriptor is not allowed for input.

Characters in a mask that have special functions are:

Z - digit selector

9 - digit selector

V - decimal alignment character

. - decimal alignment character

The field width "w" is determined by the total number of characters, including spaces but excluding V´s, between the mask delimiters. The field processed is the "w" characters starting at the current position. After the field is processed, the current position is advanced by "w" characters.

Except for the decimal point alignment character, "V", each character in the mask either defines a character position in the field or is directly inserted in the field.

The "M" edit descriptor causes numeric data elements to be rounded to the number of positions specified by the mask. String data elements are processed directly. Each digit or character of a data element is transferred to the result field in the next available character position that corresponds to a digit selector in the mask. If the digit selector is a "9", it causes the corresponding data digit to be transferred to the output field. The digit selector "Z" causes a nonzero, or embedded zero, digit to be transferred to the field, but inserts blanks in place of leading or trailing zeroes.

Character positions must be allocated, by "Z" digit selectors, within the mask to provide for the inclusion of any minus signs or decoration character strings.

A decimal point in the mask can be used for decimal point alignment of the external field. The letter "V" can also be used for this purpose. If a "V" is present in the mask, the decimal point is located at the "V", and the position occupied by the "V" is deleted. If no "V" is present, the decimal point is located at the rightmost occurrence of the decimal point character (usually "."). If neither a "V" nor a decimal point character is present, the decimal point is assumed to be to the right of the rightmost character of the entire mask.

Although leading and trailing text in a mask is always transferred to the result field, text embedded between digit selectors is transferred only if the corresponding digits to the right and left are transferred.

For example, a value that is intended to represent a date can be formatted with an "M" field descriptor as follows:

Format	Data Value	Result
M"99/99/99"	103179	|10/31/79|

The following is a comparison of the effects of using the "9" and "Z" as digit selectors. The minus sign in the preceding examples is the symbol that is automatically displayed for negative values in the absence of any specification to the contrary by a decoration. As shown in the preceding examples, a decimal point in the mask can be used for radix point alignment of the external field.

Format	Data Values	Result
3M<Z99.99>	-27.40, 12, 0	-27.40 12.00 00.00
		/\ /\ /\
3M<ZZ9.99>	-27.40, 12, 0	-27.40 12.00 0.00
		/\ /\ /\
3M<ZZZ.99>	-27.40, 12, 0	-27.40 12.00 00
		/\ /\ /\

Note: The "/\" marker is used to denote the boundaries of the output field.

In the example below, a comma specified as mask text was not displayed.

Format	Data Value	Result
M´Z,ZZ9.99´	32.009	| 32.01|

Note: The "|"character is used to denote the boundaries of the output field.

Compare the different treatment of the embedded commas in the following examples.

Data Values: 298738472, 389487.987, 666, 0.35

Format One: M<$ ZZZ,ZZZ,ZZ9 AND NO CENTS>

Format Two: M<$ 999,999,999 AND NO CENTS>

Format One	Format Two
$ 298,738,472 AND NO CENTS	$ 298,738,472 AND NO CENTS
$ 389,488 AND NO CENTS	$ 000,389,488 AND NO CENTS
$ 666 AND NO CENTS	$ 000,000,666 AND NO CENTS
$ 0 AND NO CENTS	$ 000,000,000 AND NO CENTS

The "M" edit descriptor can be useful in producing visually effective reports, by formatting values into patterns that are meaningful in terms of the data they represent. For example, assume that four arrays contain the following data:

```
Amount    := 9758 21573 15532
Date      := 031777 091779 090579
District  := ´WEST´,´MIDWEST´,´SOUTH´
Telephone := 2135296800,2162296270,4047298400
```

The following format can then be used to output the data as a table whose entries are in familiar forms. Assuming the elements are presented to the formatter in the order: the first elements of each array, followed by the second elements of each array, etc;

Using this format:

```
M<$ZZ,ZZ9>,M<   Z9/Z9/99>,3X,A8,M<   (999) 999-9999>
```

the result would be:

```
$ 9,758     3/17/77    WEST       (213) 529-6800
$21,573     9/17/79    MIDWEST    (216) 229-6270
$15,532     9/ 5/79    SOUTH      (404) 729-8400
```

MODIFIERS

Modifiers are used to alter the normal effect of edit descriptors. Modifiers immediately precede the edit descriptor to which they apply. If modifiers immediately precede the left parenthesis of a group, the modifiers apply to each repeatable edit descriptor within the group. They are enclosed in brackets, and if more than one is present, they are separated by commas.

Note: Modifiers are effective only on output. If they are supplied for input, they have no effect.

Field Blanking Modifiers

There are two modifiers for blanking fields:

```
BN = blank field if null.

BZ = blank field if equal to zero.
```

Although most edit descriptors cause a minimum number of characters to be output, a field blanking modifier causes the entire field to be filled with spaces if the specified condition is met. The null value is the value addressed by the <nullptr> in the <variablelist> entry for the current data element.

Fill Character Modifier

When an alphanumeric data element contains fewer characters than the field width specified by an "Aw" edit descriptor, when leading and/or trailing zero suppression is performed, or when embedded text in an "M" edit descriptor is not output because its neighboring digits are not, a "fill character" is inserted in each appropriate character position in the output field. The fill character is normally a space, but the fill character modifier can be used to specify any other character for this purpose.

The fill character modifier has the form:

```
    FL <char>

where

    <char> = any single character, enclosed in quotation marks
             or apostrophes.
```

The following are examples of fill character replacement:

Format	Data Value	Result
[FL'.']A10	'THEN'	|THEN.....|
[RJ,FL">"]A10	'HERE'	|>>>>>>HERE|
[FL"*"]M<$ZZ,ZZ9.99>	127.39	|$***127.39|

Note: The "|" character is used to denote the boundaries of the output field.

Overflow Character Modifier

The overflow condition occurs if there are more characters to be placed into a field than there are positions provided by the edit descriptors. In the absence of any modifier or decoration to the contrary, if an overflow condition occurs in a numeric field, the field is filled with asterisks (*). This applies to the "D", "E", "F", "G", "I", and "M" edit descriptors. The "OC" modifier can be used to substitute any other character for the asterisk as the overflow indicator character.

The "OC" modifier has the form:

```
    OC <char>

 where

    <char> = any single character, enclosed in quotation marks
             or apostrophes.
```

For example, the modifier [OC '!'] causes the output field to be filled with exclamation marks, instead of asterisks, if an overflow occurs.

Format	Data Value	Results
[OC'!']I2	100	|!!|

Note: The "|" character is used to denote the boundaries of the output field.

Justification Modifiers

The "A" edit descriptor normally displays the data left-justified in its field.

The justification modifiers are:

```
LJ - Left justify (normal)

RJ - Right justify (data is displayed right-justified)
```

The "RJ" and "LJ" modifiers are used with the "A" edit descriptor only.

Symbol Substitution Modifier

The symbol substitution modifier permits the user to replace certain standard symbols used by the formatter with symbols of his choice. It can be used with the "M" edit descriptor to free the special characters "9", "V", ".", and "Z" for use as text characters in the mask. It can also be used with the "D", "E", "F", and "G" edit descriptors to alter the standard characters they insert in the result field.

The symbol substitution modifier has the form:

```
   SS <symprs>

where

   <symprs> = one or more pairs of symbols enclosed in
              quotation marks or apostrophes.  The first
              symbol in each pair is one of those in the
              following table; the second is the symbol
              that is to temporarily replace it.
```

The following formatting symbols can be altered by the "SS" modifier:

Symbol	Function
9	Digit selector ("M" format)
Z	Digit selector, zero suppression ("M" format)
V	Decimal alignment character ("M" format)
.	Decimal point ("D", "E", "F", "G", and "M" format)

The following examples show how the "SS" modifier can be used to permit decimal values to be displayed as clock times, to follow European conventions (where a comma is used as the decimal point and periods are used as digit group separators), or to alter the function of the digit selectors in the "M" edit descriptor. When using the symbol substitution with a mask format, to obtain the function of one special character which is being altered by the symbol substitution, use the new character of the pair. With all other formats, the old character of the pair is used.

Examples:

Data Value	Format	Result
12.45	[SS".:"]F6.2	| 12:45|
12.45	[SS".:"]M<ZZZ:99>	| 12:45|
12345.67	[SS'.,']F10.2	| 12345,67|
103179	[SS<9X>]M<XX/XX/19XX>	|10/31/1979|

Note: The "|" character is used to denote the boundaries of the output field.

Table 10-1 indicates which modifiers may be used with which edit descriptors. ("Y" stands for "yes, the combination is permitted".)

Table 10-1. Modifiers Usable with Edit Descriptors

		EDIT DESCRIPTORS						
		A	E,D	F	G	I	L	M
MODIFIERS	BZ,BN	Y	Y	Y	Y	Y	Y	Y
	LJ,RJ	Y						
	OC		Y	Y	Y	Y	Y	Y
	FL	Y	Y	Y	Y	Y		Y
	SS		Y	Y	Y			Y

DECORATIONS

A decoration specifies a character string that may be added to the result field, the conditions under which the string is to be added, the location at which the string is to be added, and whether it is to be added before normal formatting is done or after it is completed.

Multiple decorations, separated by commas, may apply to the same edit descriptor. Decorations are enclosed in brackets (together with any modifiers) and immediately precede the edit descriptor to which they apply. If modifiers immediately precede the left parenthesis of a group, the modifiers apply to each repeatable edit descriptor within the group.

When a field is processed, the floating decorations appear in the same order, left to right. If an edit descriptor within a group already has some decorations, the decorations that are applied to the group function as if they were placed to the right of the decorations already present.

A decoration has the form:

```
{ M }
{ N } ... { F } <string>     OR    { M }
{ P }     { P }                    { N }
{ Z }                              { P } ... An <string>
                                   { Z }
                                   { O }
```

where

Character 1 = Field condition specifier:
- M - Minus
- N - Null
- O - Overflow
- P - Plus
- Z - Zero

Character 2 = String location specifier:
- A - Absolute
- F - Floating
- P - Prior

n = an unsigned nonzero integer constant that specifies the actual character position within the field at which the string is to begin.

<string> = any character string enclosed in quotation marks or apostrophes.

Note: Only location type "An" can be used in combination with the "O" condition.

Conditions

The condition specifier states that the string is to be added to the field if its value is minus, zero, positive, or null, or if a field overflow has occurred. A null condition takes precedence over negative, positive, and zero conditions; the overflow test is done after those for the other conditions, and therefore precedence is not significant. Alphanumeric data elements are considered to be positive or null only.

A decoration may have more than one condition specifier. If multiple condition specifiers are entered, an "or" condition is understood. For example, "ZPA2'+' " specifies that the string is to be inserted in the field if the data value is equal to or greater than zero.

Locations

The location specifier indicates where the string is to be added to the field.

The "A" specifier states that the string is to begin in absolute position "n" within the field. The leftmost position of the field is position 1.

The "F" specifier states that, once the number of data characters in the field has been established, the string is to occupy the position or positions immediately to the left (for right-justified fields) of the leftmost data character. This is reversed for left-justified elements.

The "P" specifier states that, prior to normal formatting, the string is to be inserted in the rightmost (for right-justified fields) end of the field; data characters are shifted to the left an appropriate number of positions. This is reversed for left-justified fields.

Processing

Decoration processing is as follows:

1. The data element is determined to have a negative, positive, zero, or null value; a null condition takes precedence over the other attributes.
2. If a "P" location decoration has been specified and its condition is satisfied, its string is inserted in the field.
3. Normal formatting is performed.

4. If "A" or "F" decorations have been specified and their conditions met, they are applied.

5. If an attempt has been made to transfer more characters to the field than can be accommodated (in step 2, 3, or 4), the overflow condition is set. If an overflow decoration has been specified, it is applied.

Note: Only location type "An" can be used with the "O" condition.

Examples:

Format	Data Value	Result
[MF'<',MP'>',ZPP' ']F12.2	1000.00	1,000.00
[MF'<',MP'>',ZPP' ']F12.2	-1000.00	<1,000.00>
[MA1'CR',MPF'$']F12.2	1000.00	$1,000.00
[MA1'CR',MPF'$']F12.2	-100.00	CR $100.00
[OA1<**OVERFLOW**>]F12.2	1000000.00	1,000,000.00
[OA1<**OVERFLOW**>]F12.2	10000000.00	**OVERFLOW**

Note: The "|" character is used to denote the boundaries of the output field.

The following decorations are automatically applied to any numeric edit descriptor ("D", "E", "F", "G", "I", or "M") for which no decoration has been specified:

```
MF'-'
OA1'*** ... *'  (The number of asterisks is equal to the
                 number of characters in the field width.)
```

However, if any decoration with a condition code relating to the sign of the data is specified, the automatic "MF'-'" decoration no longer applies; if negative-value indication is desired, the user must supply the appropriate decoration. If any decoration with a condition code relating to overflow is specified, the automatic "OA1'***...*'" decoration no longer applies.

As an example of how decorations apply to a group of edit descriptors, the following formats give the same results:

Format

```
[MF'-'](F10.2,[MZF'**']F10.2)

[MF'-']F10.2,[MZF'**',MF'-']F10.2
```

Using the format above:

Data Values	Results
0,0	0.00 **0.00
	/\ /\ /\
1,1	1.00 1.00
	/\ /\ /\
-1,-1	-1.00 **-1.00
	/\ /\ /\

Note: The "/\" marker is used to denote the boundaries of the output field.

LIST-DIRECTED FORMATTING

List-directed formatting provides the data conversion capabilities of the formatter without requiring the specification of a format. The FORMATDATA procedure determines the details of the data conversion, based on the types of the data elements. This is particularly convenient for input, because the list-directed formatting rules provide for free format input of data values rather than requiring data to be supplied in fixed fields. There are fewer advantages to using list-directed formatting for output, because the output data is not necessarily arranged in a convenient readable form.

The characters in one or more list-directed buffers constitute a sequence of "data list items" and "value separators". Each value is either a constant, a null value, or one of the forms:

```
        r*c                     r*

  where

     r is an unsigned, nonzero, integer constant.

     r*c form is equivalent to "r" successive appearances of the
     constant "c".

     r* form is equivalent to "r" successive null values.

     Neither of these forms may contain embedded blanks, except
     where permitted with the constant "c".
```

List-Directed Input

All input forms which are acceptable to FORMATDATA when directed by a format are acceptable for list-directed input, with the following exceptions:

1. When the data element is a complex variable, the input form consists of a left parenthesis followed by an ordered pair of numeric input fields separated by a comma, and followed by a right parenthesis.

2. When the data element is a logical variable, the input form must not include either slashes or commas among the optional characters for the "L" editing.

3. When the data element is a character variable, the input form consists of a string of characters enclosed in apostrophes. The characters blank, comma, and slash may appear in the string of characters.

4. A null value is specified by having no characters other than blanks between successive value separators, no characters preceding the first value separator in the first buffer, or the r* form. A null value has no effect on the value of the corresponding data element. The input list item retains its previous value. A single null value must represent an entire complex constant (not just part of it).

If a slash value separator is encountered during the processing of a buffer, data conversion is terminated. If there are additional elements in the data list, the effect is as if null values had been supplied for them.

On input, a value separator is one of the following:

1. A comma or slash optionally preceded or optionally followed by one or more contiguous blanks (except within a character constant).

2. One or more contiguous blanks between two constants or following the last constant (except embedded blanks surrounding the real or imaginary part of a complex constant).

3. The end of the buffer (except within a character constant).

List-Directed Output

Output forms that are produced by list-directed output are the same as that required for input with the following exceptions:

1. The end of a buffer may occur between the comma and the imaginary part of a complex constant only if the entire constant is as long as, or longer than, an entire buffer. The only embedded blanks permitted within a complex constant are between the comma and the end of a buffer, and one blank at the beginning of the next buffer.

2. Character values are displayed without apostrophes.

3. If two or more successive values in an output record produced have identical values, the FORMATDATA procedure produces a repeated constant of the form "r*c" instead of the sequence of identical values.

4. Slashes, as value separators, and null values are not produced by list-directed output.

For output, the value separator is a single blank. A value separator is not produced between or adjacent to character values.

SECTION 11

COMMAND INTERPRETER/APPLICATION INTERFACE

The following topics are covered in this section:

- General Characteristics of the Command Interpreter
- File Names
  - External Form
  - File Name Expansion
  - Default Volume and Subvol Names
- How Parameter Information is Passed to an Application Process
  - RUN Command
    - Startup Parameter Message
  - ASSIGN Command
    - Logical File Assignment Messages
  - PARAM Command
    - Param Messages
- Application Process to CI Interprocess Messages
  - Display Message
  - Wakeup Message
- Application-supplied CI Monitor Process
  - LOGON/LOGOFF Message
  - RUN Command Message

GENERAL CHARACTERISTICS OF THE COMMAND INTERPRETER

The Command Interpreter provides a direct interface between system users (people) and the GUARDIAN operating system. Users at on-line terminals interact with the Command Interpreter by typing in commands. If a command is given to run a program, then the program begins executing. If a command is given to perform some specific operation, the necessary system functions are executed. The Command Interpreter, in the latter case, then asks the user for another command (by displaying a colon ":" on the terminal).

Some functions that the Command Interpreter performs are:

- List disc file names
- Create, rename, and purge disc files

- Set default disc volume and subvol names and default security
- Run and pass parameters to processes
- Put a process into the debug state
- Stop process execution

Most of these functions do not directly affect application program design. The programmer, however, must be aware of how the Command Interpreter passes parameters to application processes and how the default volume and subvol names are used. (For a user description of all Command Interpreter functions, see the GUARDIAN Operating System Command Language and Utilities Manual).

An additional consideration is that the Command Interpreter makes use of the BREAK feature on the home terminal. Because of this, any application process that is run via the Command Interpreter and also uses BREAK on the home terminal, must do so in a proper manner. See "Using BREAK (more than one process per terminal)" in the conversational/page mode terminal part of the "File Management" section in this manual.

For a discussion of default security, see section 7, "Security System".

FILE NAMES

Many Command Interpreter commands require that a file name be specified. A file name can be that of a disc device, non-disc device, or named process. The file name of a non-disc device is represented to the Command Interpreter in the same form as the file system's internal representation; that is:

$<device name> or

$<logical device number>

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

$<volume name>.<subvol name>.<disc file name>

example:

$STORE1.ACCTRCV.SORTFILE

A name of a process is similar to a disc file name in that it can consist of three parts with each part separated from the other by a period. Unlike a disc file name, however, only the left-hand part is required. This yields a process name of the following form:

$<process name>[.#<1st qualif name>[.<2nd qualif name>]]

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

Correspondence of External to Internal File Names

The correspondence of the external form of file names to the internal form is shown in figure 11-1.

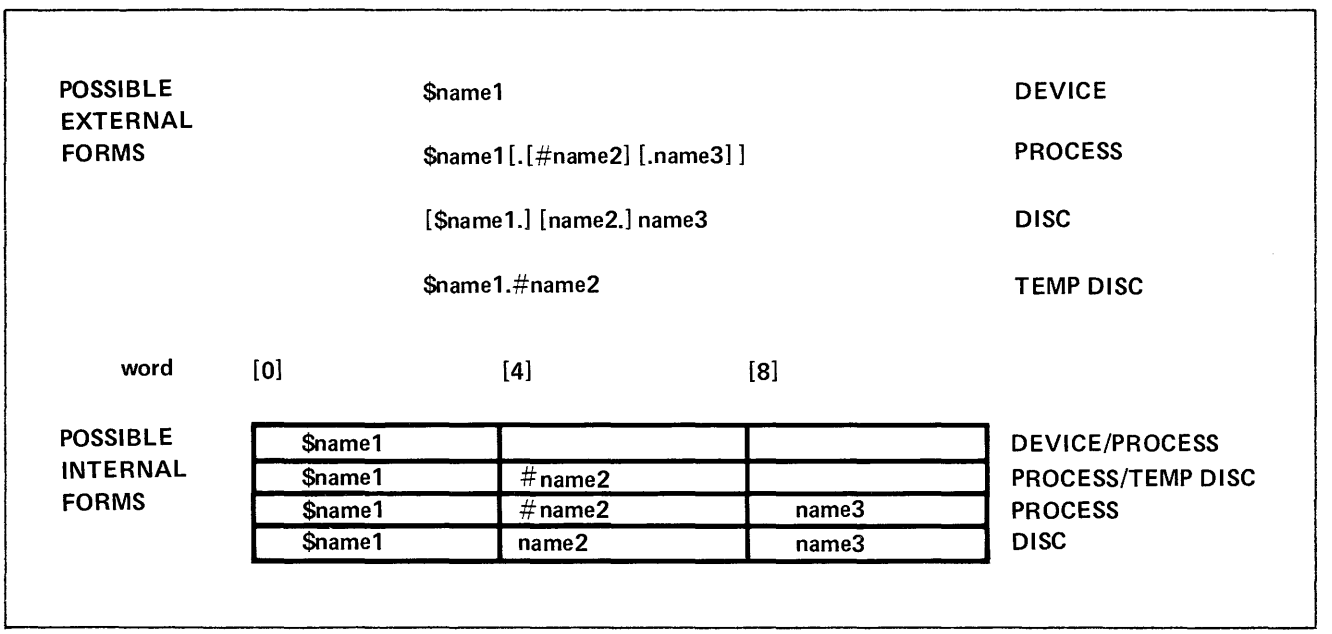


Figure 11-1. File Names

When the external form of a file name is entered as a parameter to a command (e.g., the IN parameter of the Command Interpreter RUN command), it is converted to the internal form as shown above. For example, in the case of the IN parameter <file name>, the <file name> is converted from the external form to the internal form (and expanded if necessary) and sent to the application process in an interprocess "startup" message.

Disc File Name Expansion

As an operating convenience, the Command Interpreter accepts, where a file name is a parameter to a command, disc file names of the following forms (see figure 11-2):

- <disc file name>
- <subvol name>.<disc file name>
- $<volume name>.<disc file name>
- $<volume name>.<subvol name>.<disc file name>
- <temporary file name>
- $<volume name>.<temporary file name>
- Network file names (see EXPAND User's Manual)

When a partial file name is supplied, the internal representation of the file name is expanded into a full three-part file name. This is accomplished as follows: There is a "default volume name" and a "default subvol name" associated with each Command Interpreter. A partial name is expanded by merging the default volume and/or subvol names into the omitted part(s) of the partial file name. As a minimum, a partial file name must consist of a <disc file name>. Partial file names are expanded into full file names according to the following rules:

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

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

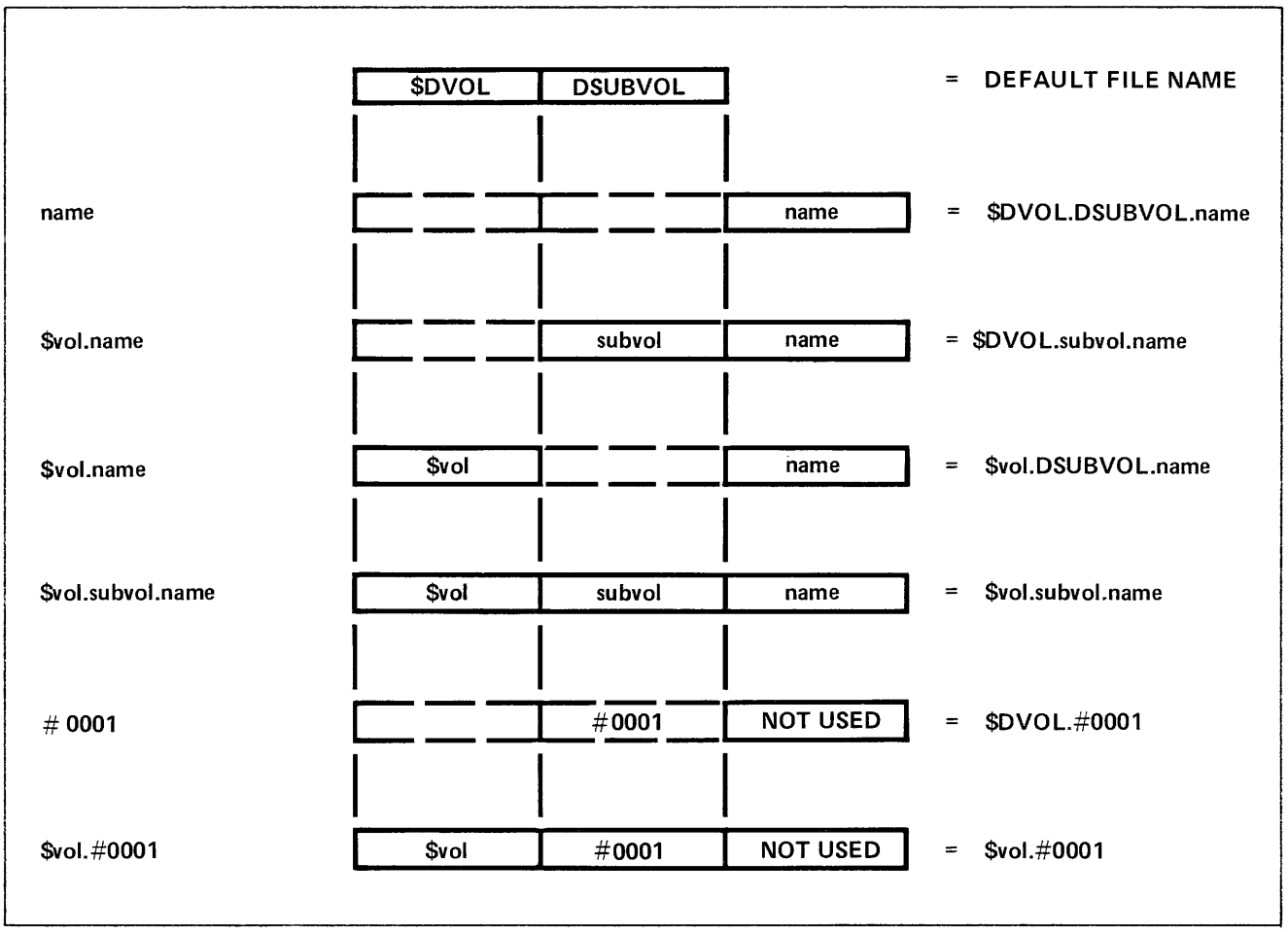


Figure 11-2. Disc File Names

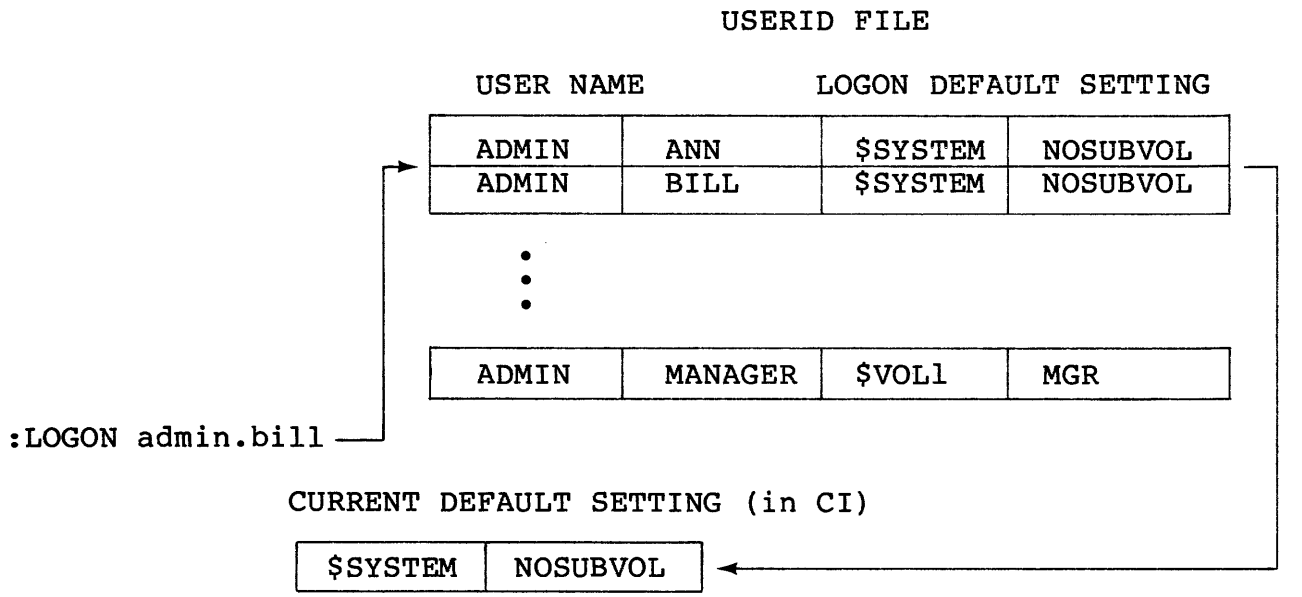
HOW THE DEFAULT FILE NAMES ARE ESTABLISHED. Application-predefined default file names are automatically established for a given user at logon time (i.e., when the Command Interpreter LOGON command is given). The application-predefined names are referred to as the "logon default setting". Each user's logon default setting is kept in the USERID file (the USERID file contains the names of all users defined for the system).

The default file names established for a user at logon time are referred to as the "current default setting". The current default setting is kept by the Command Interpreter and used for file name expansion.

The first time a new user logs on, the user's logon default setting is

"$SYSTEM.NOSUBVOL"

For example, the first time the user "admin.bill" logs on after he has been established as a system user, the following takes place:



The Command Interpreter reads the USERID file entry for "ADMIN.BILL". The logon default setting, "$SYSTEM.NOSUBVOL" is used to establish the Command Interpreter's current default setting.

Following LOGON, a user can, via the DEFAULT command, change the default names to be in effect for future logons. The DEFAULT command has three forms:

DEFAULT $<default volume name>.<default subvol name>

sets new logon defaults for both volume and subvol names.

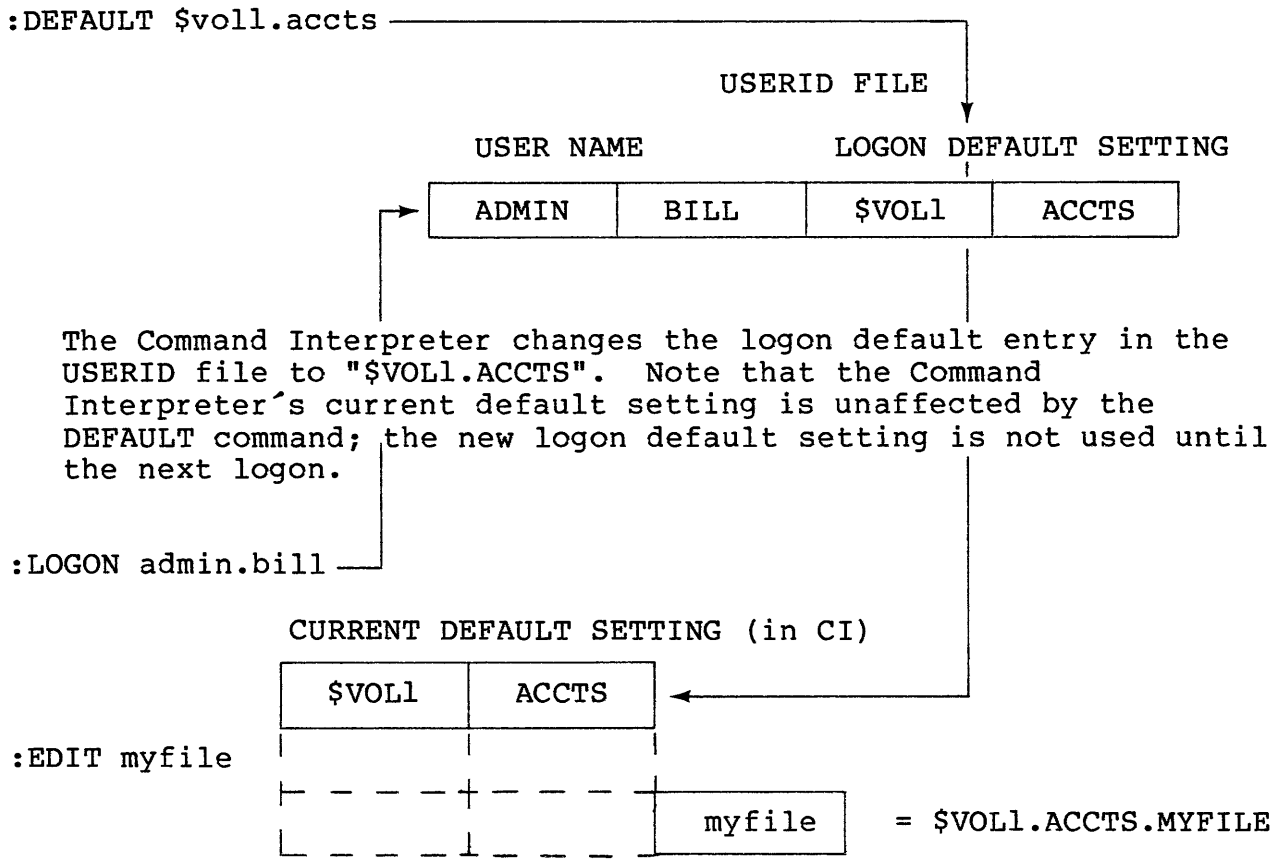
DEFAULT $<default volume name>

sets a new logon default name for the volume name; the logon default subvol name is unchanged.

DEFAULT <default subvol name>

sets a new logon default name for the subvol name; the logon default volume name is unchanged.

For example:



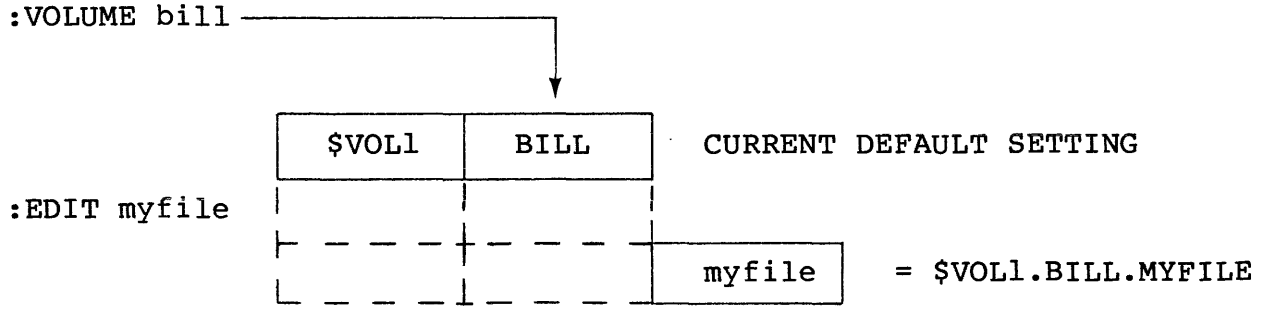
To change the "current default setting" following a logon (without affecting the "logon default setting"), the VOLUME Command is used. The VOLUME command has three forms which are equivalent to those of the DEFAULT command:

```
VOLUME $<default volume name>.<default subvol name>
VOLUME $<default volume name>
VOLUME <default subvol name>
```

For example, for the user "admin.bill" who has the current default setting

```
$VOL1.ACCT
```

wants to change the current default setting "$voll.bill" without affecting the logon default settings. Therefore, the following VOLUME command is entered:




```
 T9601B04  EDIT (780113)
 CURRENT FILE IS $VOL1.BILL.MYFILE
*
.
```

The use of default names has the effect of automatically placing all files for a particular user on the desired volume and assigning the desired subvol name. This has the effect of keeping the files of various users separated.

Note: The default volume and subvol names are sent to the application process as part of the Command Interpreter startup message.

Network File Names

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

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

The external form of a network file name is:

```
\<system name>.<external file name>

where

   <external file name>

      is the external form of any legal file name.  Note that the
      length of a device or process name used with a system name
      contains one character less than usual:  device names have
      at most six alphanumerics, and process names have at most
      four alphanumerics.

examples:

   \newyork.$system.system.myfile  ! fully qualified
                                   ! disc file name

   \remote.$xyz  ! process

   \detroit.#1234  ! temporary disc file name
```

DEFAULT SYSTEM. Each Command Interpreter running on a system in a network has associated with it a default system that is used in file name expansion. When a user logs on, the default system is always the system on which the user's Command Interpreter is running. The default system can be changed via the Command Interpreter SYSTEM command.

EXPANSION OF NETWORK FILE NAMES. File names presented in external form to a Command Interpreter (or any other Tandem subsystem, such as the Editor or FUP) running on a system in a network are expanded using the default volume, subvolume, and system names. Some examples, assuming that the current user's defaults are as follows:

```
default volume:     $myvol
default subvolume:  mysubvol
default system:     \calif
```

this file name presented to Command Interpreter		is expanded to
myfile	⟶	\calif.$myvol.mysubvol.myfile
\newyork.myfile	⟶	\newyork.$myvol.mysubvol.myfile
$proc	⟶	\calif.$proc

CORRESPONDENCE OF INTERNAL TO EXTERNAL NETWORK FILE NAMES. When transforming an external file name to an internal one, the system replaces the system name with the corresponding system number. External network file names supplied as IN or OUT files in a RUN command are converted to internal form by the Command Interpreter. Thus an application process that reads its startup message and opens its IN file need not do anything different when remote files are involved.

PASSING RUN-TIME PARAMETER INFORMATION TO AN APPLICATION PROCESS

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

There are five Command Interpreter commands that can affect the parameter information to be sent. They are:

1. The VOLUME command specifies the default volume and subvolume names and default security to be passed to the new process.

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

   The default volume and subvol names, the IN and OUT file names, and the optional parameter string are passed to the application process in the startup message.

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

4. The PARAM command is used to associate an ASCII value with a parameter name. This command is typically used by languages such as COBOL or FORTRAN to give initial values to program variables. If any PARAMs are in effect when a program is run, a "param" message containing the parameter names and values is sent, at the option of the new process. This follows the transmission of any assign message(s).

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

Note: If a process opens the $RECEIVE file and specifies that it wishes to receive OPEN, CONTROL, SETMODE, and CLOSE system messages, the first message it receives will be an OPEN message. This will be followed by the parameter message(s), then followed by a CLOSE message (the OPEN and CLOSE messages are caused by the Command Interpreter opening and closing the file to the new process).

RUN Command

The RUN command is used to run programs. The program's input and output files, processor module, execution priority, number of data pages, and process name can optionally be specified.

There are two forms of the RUN command: implicit and explicit. The implicit form is assumed by the Command Interpreter if a command is input that it does not recognize. The explicit form is used when the command RUN[D] is entered.

The form of the RUN command is:

```
[RUN[D]] <file name> [ / <parameters> / ] [ <parameter string> ]
         -----------
```

where

[RUN[D]]

omitting "RUN" is an implicit RUN command. The Command Interpreter attempts to run a program located in $SYSTEM.SYSTEM.<file name>.

including "RUN" or "RUND" (debug) is an explicit RUN command. A partial <file name> is expanded as previously described.

<parameters> are

IN [<filename>]

is the new process's input file. <file name> is expanded and sent to the new process in the parameter message. If "IN <file name>" is omitted, the file name of the Command Interpreter's IN file is sent (usually, this is the logical device number of the home terminal). If "IN" is included but <file name> is omitted, blanks are sent as the name of the input file.

OUT [<filename>]

is the new process's output file. <file name> is expanded and sent to the new process in the parameter message. If "OUT <file name>" is omitted, the file name of the Command Interpreter's OUT file is sent (usually, this is the <logical device number> of the home terminal). If "OUT" is included but <file name> is omitted, blanks are sent as the name of the output file.

→

NAME [<process name>]

is the symbolic name to be assigned to the new process. If this parameter is omitted, the process is unnamed. If "NAME" is included but <process name> is omitted, the system generates a name for the new process. The process's name is entered into the Process-Pair Directory (PPD).

CPU <cpu num>

{0:15}, is the processor module where the new process is to execute. If omitted, the same processor module as the Command Interpreter's is assigned.

PRI <priority>

{1:199}, is the execution priority of the new process. If omitted, a priority of one less than the Command Interpreter's is assigned. If a value greater than 199 is specified, the process is run at priority 199.

MEM <num pages>

{1:64}, is the maximum number of virtual data pages to be allocated the new process. If omitted or less than the number assigned at compilation time, the compilation value is used.

NOWAIT

if specified, means that the Command Interpreter does not pause (i.e., suspend itself) when the program is run. Instead, it returns with a command input prompt as soon as the new process reads its startup message. Normally this keyword is not specified (if NOWAIT is not specified, the Command Interpreter pauses when a program is run).

TERM <terminal name>

specifies the terminal to be used for the new process's home terminal. If omitted, the Command Interpreter's home terminal is used.

<parameter string>

is sent to the new process in the startup parameter message. Leading blanks are ignored.

→

```
example:

   :VOLUME $storel.acctrcv
   :RUN xnstp /IN infile,OUT outfile,NAME/ 1,10,byte,descending
       ! runs "$storel.acctrcv.xnstp"
```

Startup Message

The startup message is sent to the new process immediately following the successful creation of the new process. The startup message is read by the process via its $RECEIVE file.

The form of the startup parameter message is:

```
STRUCT ci^startup;
  BEGIN                          !  word
    INT msgcode;                 !  [0] -1.
    STRUCT default;
      BEGIN
        INT volume [ 0:3 ], !    [1] $<default volume name>.
            subvol [ 0:3 ]; !        <default subvol name>.
      END;
    STRUCT infile;
      BEGIN
        INT volume [ 0:3 ], !    [9] IN parameter <file name> of RUN
            subvol [ 0:3 ], !        command.
            dname  [ 0:3 ];
      END;
    STRUCT outfile;
      BEGIN
        INT volume [ 0:3 ], ! [21] OUT parameter <file name> of RUN
            subvol [ 0:3 ], !        command.
            dname  [ 0:3 ];
      END;
    STRING param [ 0:n-1 ]; ! [33] <parameter string> of RUN
  END; ! ci^startup^msg.     !      command (if any) that was
                             !      entered by operator.  This is in
                             !      either of the following forms:
                             !
                             !      <parameter string><null>[<null>]
                             !
                             !      or
                             !
                             !      <null><null>
                             !
                             !      <n> = ( <count read> - 66 ) / 2
```

The maximum length possible for a startup message is 596 bytes (including the trailing null characters).

Note: The parameter message length is always an even number. If necessary, the Command Interpreter pads the <parameter string> with an additional null.

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

First, the following VOLUME command is entered:

```
:VOLUME $storel.acctrcv
```

Then the following RUN command is given:

```
:RUN xnstp/IN infile, OUT outfile, NAME/ 1,10,byte,descending
```

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

The Command Interpreter forms a message to be sent to the new process from the IN parameter information, the OUT parameter information, and the application-dependent parameter string. The message contains the following information:

```
word[0]  = - 1                              ! means start-up message.
word[1]  = "$STOREl "                       ! $<default volume name>.
word[5]  = "ACCTRCV "                       ! <default subvol name>.
word[9]  = "$STOREl ACCTRCV INFILE  "       ! IN param <file name>.
word[21] = "$STOREl ACCTRCV OUTFILE "       ! OUT param <file name>.
word[33] = "1,10,byte,descending"           ! <parameter string>.
word[43] = <null><null>                     ! null terminator(s).
```

The Command Interpreter then attempts to run the program indicated by the expanded form of "xnstp" - $storel.acctrcv.xnstp. If the new process is created, it is sent its startup parameter message.

The first action the "xnstp" program performs is to open and read the $RECEIVE file:

```
INT .receive[0:11] := ["$RECEIVE", 8 * ["  "]], ! data declarations
     recv^fnum,
     in^fnum,
     out^fnum,
     num,
    .buffer[0:99],
     num^read,
    .creator[0:3],
    .lastpid[0:3];

STRING .parms[0:39],
       .sbuffer := @buffer '<<' 1;
```

```
   .
CALL OPEN(receive, recv^fnum);
   .
CALL READ(recv^fnum, buffer, 200, num^read);
   .
```

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

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

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

```
   .
CALL OPEN(buffer[9],in^fnum);
   .
   opens "$storel acctrcv infile"

   .
CALL OPEN(buffer[21], out^fnum);
   .
   opens "$storel acctrcv outfile"
```

then saves the <parameter string> information:

```
   .
num := num^read - 66; ! length of parameter string in bytes.
IF num <= 40 THEN     ! parameter string will fit. move it in.
   parms ':=' sbuffer[66] FOR num
ELSE                  ! parameter string too long.
   .
```

ASSIGN Command

The ASSIGN command is used to make or display logical file assignments for programs written in such languages as COBOL and FORTRAN. A logical file assignment equates a Tandem file name (as described under "File Names" in the "File Management System" section of this manual) with a file in a program and, optionally, assigns characteristics to that file.

The form of the ASSIGN command is:

```
ASSIGN [ <logical unit> [ , [ <Tandem file name> ]
------
       [ { , <create-open spec> } ...  ] ] ]
```

where

<logical unit>

is the name to be assigned a file attribute. <logical unit> is of the form

```
[ { <program unit> } . ] <logical file>
  {       *       }
```

<program unit> and <logical file> consist of one to thirty-one alphanumeric, caret "^", or hyphen "-" characters.

The exact meaning of <program unit>, the literal *, and <logical file> depends on the application (e.g., the COBOL or FORTRAN object program). In general, <program unit>, if present, is the name of the program unit (as given in the source program itself) to which the file name assignment is to apply; *, if present, means apply the assignment to all program units in the object program file being run; and <logical file> is the name of the file as given in the source program (for FORTRAN, commonly FTnnn, where nnn is the unit number). <program unit> and <logical file> have no meaning for FORTRAN. For details on FORTRAN or COBOL application treatment of ASSIGNs and PARAMs, see the appropriate language manual. TAL programmers must access assign and param messages explicitly.

If <logical unit> is omitted, the Command Interpreter displays the assigned values for all assignments currently in effect.

→

<Tandem file name>

is a file name in the external form. A partial file name is not expanded. However, the application process can expand the file name using the default information passed in the startup message.

If <Tandem file name> is omitted and <create spec> is omitted, the current assignment value for <logical unit> is displayed.

If <Tandem file name> is omitted but <create spec> is supplied, blanks are passed in the <Tandem file name> field of the assign message.

<create-open spec>

are one or more file creation or open attributes. <create-open spec> is of the form:

```
{ <extent spec>      }
{ CODE <file code>   }
{ <exclusion spec>   }
{ <access spec>      }
{ REC <record size>  }
{ BLOCK <block size> }
```

<extent spec> is

```
{ EXT [ ( ] <pri extent size> [ ) ]                  }
{ EXT ( [ <pri extent size ] , <sec extent size> ) }
```

<exclusion spec> is

```
{ EXCLUSIVE }
{ SHARED    }
{ PROTECTED }
```

<access spec>

```
{ I-O    }
{ INPUT  }
{ OUTPUT }
```

<record size>, <block size>, and <extent size> are integers in the range of {0:65535}.

→

```
examples:

   ! assign Tandem file name and creation attributes (COBOL file).
   ASSIGN print-file, myfile, EXT 16, CODE 9999, EXCLUSIVE, OUTPUT
   ! assign Tandem file name and creation attributes (FORTRAN file).
   ASSIGN FT002, datafile, INPUT, EXCLUSIVE
   ! assign create-open attributes to the default file.
   ASSIGN print-file,,SHARED
   ! display the assigned attributes of a designated logical file.
   ASSIGN print-file
   ! display the assigned attributes of all logical files.
   ASSIGN
```

CONSIDERATIONS

- A maximum of 15 assignments can be in effect at one time. This number can be increased by increasing the Command Interpreter's default data space of 8 pages. Increasing the Command Interpreter's data space requires the use of the MEM option when the Command Interpreter is run:

  ```
  :COMINT /. . .,MEM 9, . . ./
  ```

  The command just shown starts a new Command Interpreter with 9, rather than 8, data pages.

- Notice that the Command Interpreter only stores the values assigned by this command and sends the values to requesting processes, at process startup time, in the form of assign messages. This command does not create files; the interpretation of the assigned values must be made by application programs.

- The Command Interpreter creates an assign message for each ASSIGN in effect. A new process must request its assign messages (if any) following receipt of the startup message. The COBOL and FORTRAN compilers provide the code for this function. TAL programs that use assigns must provide their own code for handling assign messages.

- To delete existing assigns, use the CLEAR command.

Assign Message

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

- replies to the startup message with an error return value of REPLY = 70. The Command Interpreter then sends both assign and param messages.

- replies to the startup message with an error return value of 0, but with a reply of one to four bytes, and bit 0 of the first byte of the reply is set to 1. The Command Interpreter also sends param messages if bit 1 of the first byte of the reply is set to 1.

The form of the assign message is:

```
STRUCT ci^assign;                 ! assign message.
  BEGIN                           !
    INT msg^code;                 !  [0] -2
                                  !
    STRUCT logicalunit;           ! PARAMETERS TO ASSIGN COMMAND.
      BEGIN                       !
        STRING prognamelen,       !  [1] length in bytes of name {0:31}
               progname[0:30],    !      { <program unit> | * }<blanks>
               filenamelen,       ! [17] length in bytes of name {0:31}
               filename[0:30];    !      <logical file><blanks>
      END;                        !

    INT(32) fieldmask;            ! [33] bit mask to indicate which of
                                  !      the following fields were
                                  !      supplied (1 = supplied):
                                  !      .<0> = <Tandem file name>
                                  !      .<1> = <pri extent size>
                                  !      .<2> = <sec extent size>
                                  !      .<3> = <file code>
                                  !      .<4> = <exclusion size>
                                  !      .<5> = <access spec>
                                  !      .<6> = <record size>
                                  !      .<7> = <block size>
                                  !
    STRUCT tandemfilename;        ! [35] <Tandem file name>
      BEGIN                       !
        INT volume  [ 0:3 ],      !
            subvol  [ 0:3 ],      !
            dfile   [ 0:3 ];      !
      END;                        !
    ! createspec                 !
    INT primaryextent,            ! [47] <pri extent size>.
        secondaryextent,          ! [48] <sec extent size>.
        filecode,                 ! [49] <file code>.
        exclusionspec,            ! [50]   %00 if SHARED,     } corre-
                                  !        %20 if EXCLUSIVE,  } sponds
                                  !        %60 if PROTECTED.  } to flag
        accessspec,               ! [51] %0000 if I-O,        } param of
                                  !      %2000 if INPUT,      } OPEN.
                                  !      %4000 if OUTPUT.     }
        recordsize,               ! [52] <record size>.
        blocksize;                ! [53] <block size>.
  END;
```

The length of this message is 108 bytes.

PARAM Command

The PARAM command is used to assign a string value to a parameter name, or to display such assignments. As a specific example, the PARAM command is used to pass status switch settings to COBOL programs. However, this command is available for use with other languages as well.

The form of the PARAM command is:

```
PARAM [ { <parameter name> <parameter value> } , ... ]
-----
```

where

<parameter name>

is the name to be assigned a <parameter value>. <parameter name> consists of one to thirty-one alphanumeric, caret "^", or hyphen "-" characters.

<parameter value>

is the value to be assigned to <parameter name>. <parameter value> may have either of the following forms:

```
{ <character string>   }
{ "<character string>" }
```

If the first form is used, the string cannot contain any embedded commas ","; leading and trailing blanks are not significant (i.e., are not included as part of the <parameter value>).

If the second form is used, all characters, including leading and trailing blanks between the quotation marks, are significant (i.e., they are included as part of the <parameter value>). An embedded quotation mark is represented by a pair of quotation marks "".

If { <parameter name> <parameter value> } is omitted, the assigned values for all parameters currently in effect are displayed.

→

examples:

```
! assign a value to two parameters in a COBOL program.
PARAM SWITCH-1 ON, DEBUG ON
! assign a value to a parameter in a FORTRAN program.
PARAM BACKUPCPU 3
! assign a value to a parameter.
PARAM string " a string with an embedded quote "" "
! display the assigned values for all parameters.
PARAM
```

CONSIDERATIONS

- Notice that the Command Interpreter only stores the values assigned by the PARAM command and sends the values to requesting processes, at process startup time, in the form of param messages. The interpretation of the assigned values must be made by the application program.

- The Command Interpreter's internal storage for params is 1024 bytes. Therefore, the maximum number of params that can be in effect is limited by this value. Each param uses

  2 + <parameter name length> + <parameter value length>

  bytes.

- To delete existing params, use the CLEAR command.

Param Message

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

- replies to the startup message with an error return value of REPLY = 70. The Command Interpreter then sends both assign and param messages.

- replies to the startup message with an error return value of 0, but with a reply of one to four bytes, and bit 1 of the first byte of the reply is set to 1. The Command Interpreter also sends assign messages if bit 0 of the first byte of the reply is set to 1.

The form of the param message is:

```
STRUCT ci^param;                    ! param message.
  BEGIN                             !
    INT msg^code,                   !  [0] -3
        numparams;                  !  [1] number of parameters
                                    !      included in this message.
    STRING parameters [ 0:1023 ]; !  [2] beginning of parameters.
  END;
```

The field "parameters" in the above message format is comprised of "numparams" records of the form (offsets are given in bytes):

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

The maximum length of this message is 1028 bytes.

CLEAR Command

The CLEAR command is used to clear references set by the ASSIGN and PARAM commands.

The form of the CLEAR command is:

```
        { ALL [ { ASSIGN | PARAM } ] }
        { ASSIGN <logical unit>      }
CLEAR   { PARAM <parameter name>     }
```

where

ALL

means clear all currently assigned ASSIGN and PARAM references.

ALL ASSIGN

means clear all currently assigned ASSIGN references.

ALL PARAM

means clear all currently assigned PARAM references.

ASSIGN <logical unit>

means clear the reference to <logical unit>.

PARAM <parameter name>

means clear the reference to <parameter name>.

examples:

```
! clear all assigns and params.
CLEAR ALL
! clear all assigns.
CLEAR ALL ASSIGN
! clear assignment for "print-file".
CLEAR ASSIGN print-file
! clear parameter "SWITCH-1".
CLEAR PARAM SWITCH-1
```

Reading All Parameter Messages

If it is desired to read all parameter messages, the following must be taken into consideration:

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

   OPEN <receive depth> >= 1

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

   OPEN <flags>.<1> = 1

   so that it will receive OPEN and CLOSE system messages.

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

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

2. Startup message (message code = -1)

   Application process must reply with REPLY <error return> = 70.

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

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

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

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

```
PROC read^parameter^messages;
  BEGIN
    INT .rcv^fname [ 0:11] := [ "$RECEIVE", 8 * [ "   " ] ],
         rcv^fnum,
        .rcv^buf   [ 0:514 ]
         cnt^read,
         reply^code        := 70;

    LITERAL
         rcv^flags  = %40000, ! OPEN-CLOSE messages.
         rcv^depth  =     1, ! READUPDATE-REPLY.
         rcv^cnt    =  1030, !
         close^msg  =   -31; ! CLOSE message code.

    ! open $RECEIVE.
    CALL OPEN ( rcv^fname , fcv^fnum , rcv^flags , rcv^depth );
    IF <> THEN ...;
    ! read open message.
    CALL READUPDATE ( rcv^fnum , rcv^buf , rcv^cnt , cnt^read );
    WHILE rcv^buf <> close^msg DO
      BEGIN
        CASE $ABS ( rcv^buf ) OF
          BEGIN
    !    0 ! ;
    !   -1 ! BEGIN ! startup message.
                    .
                    process startup message.
                    .
              END;
    !   -2 ! BEGIN ! assign message.
                    .
                    process assign message.
                    .
              END;
    !   -3 ! BEGIN ! param message.
                    .
                    process param message.
                    .
              END;
              OTHERWISE;
          END;
        CALL REPLY ( ,,,, reply^code );
        CALL READUPDATE ( rcv^fnum, rcv^buf, rcv^cnt, cnt^read);
      END; ! while not close^msg.
    ! close $RECEIVE.
    CALL REPLY (,,,,0); ! reply to close^msg.
    CALL CLOSE ( rcv^fnum );
  END; ! read^parameter^messages.
```

APPLICATION PROCESS TO CI INTERPROCESS MESSAGES

There are two messages which the Command Interpreter accepts from application processes:

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

An interprocess message is sent to a particular Command Interpreter by opening a process file to that Command Interpreter, then writing the message via the WRITE procedure (see "Interprocess Communication" in the "File Management" section).

Wakeup Message

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

If the Command Interpreter is not in the pause state (i.e., it is prompting for a command or executing a command other than RUN), a wakeup message is ignored.

The form of the wakeup message is:

```
STRUCT wakeup^msg;
  BEGIN
    INT msgcode; ! -20
  END;
```

The length of this message is two (2) bytes.

The intended use of this message is to allow a process that is a descendant of a Command Interpreter, but not a direct descendant, to wake up that Command Interpreter. A typical case is with a non-named process pair: the backup process calls STEPMOM with the primary process the object of the call (so that the backup will know if the primary fails); the call to STEPMOM cancels the primary process's relationship with the Command Interpreter. The primary process, just prior to stopping, sends a wakeup message to the Command Interpreter.

Display Message

The display message, when received by a Command Interpreter, causes the Command Interpreter to display the text contained in the message. The text is displayed just prior to the next time the Command Interpreter prompts for a command (i.e., issues a ":").

A Command Interpreter has the capability to store one 132-byte display message until it is able to display the message text. If the Command Interpreter is currently storing a display message when another display message is sent to it, the second display message is rejected with an <error> 12 indication (file in use).

The form of the display message is:

```
STRUCT display^msg;
  BEGIN
    INT msgcode;            ! -21
    STRING text [ 0:n-1 ]; ! n <= 132.
  END;
```

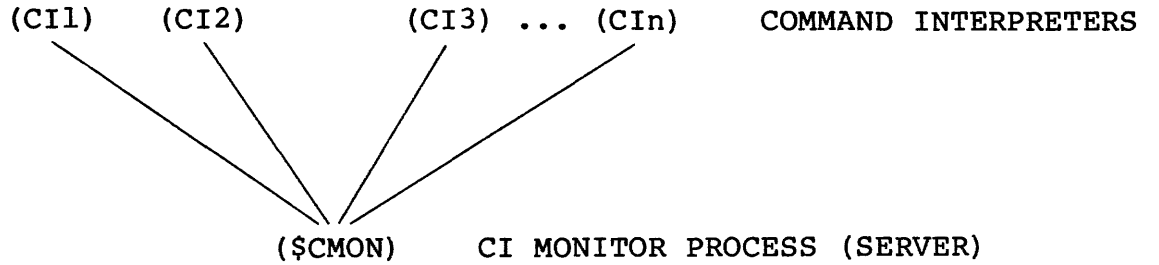
The length of this message is (2 + display text length) bytes. Note that the length of the text portion is implied in the write count used to send this message.

APPLICATION-SUPPLIED CI MONITOR PROCESS ($CMON)

The purpose of the application-defined CI Monitor Process is to allow application-dependent control of

- logons and logoffs
- running of programs

To provide this control, the Command Interpreter attempts to open a process file to a process named $CMON each time a LOGON command is executed. If the open is successful (i.e., a $CMON process exists), the Command Interpreter notifies the $CMON process each time a LOGON, LOGOFF, or implicit or explicit RUN command is given (this notification is made in the form of an interprocess message):



The relationship between all Command Interpreters in the system and the $CMON process is that of requestors and server, respectively. (See "Interprocess Communication" in the "File Management" section.)

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

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

Communication between Command Interpreters and $CMON

The interprocess messages from a Command Interpreter to the $CMON process are:

- Logon Message

- Logoff Message

- Process Creation Message (i.e., RUN command)

All messages are sent to $CMON by Command Interpreters on a no-wait basis. If a message cannot be sent or if $CMON does not reply, the Command Interpreter closes the process file to $CMON and proceeds to execute the command, as the $CMON process does not exist.

If a non-super ID user presses the BREAK key while a message is outstanding to $CMON, the message is cancelled and the command is aborted. If a super ID user presses the break key while a message is outstanding, the message is cancelled and the command is executed.

If the Command Interpreter encounters an i/o error when communicating with $CMON, it closes its file to $CMON and no longer attempts communication. The Command Interpreter reopens $CMON (and again attempts communication) when the next logon occurs.

LOGON MESSAGE: This message is sent to the $CMON process when a LOGON command is entered and the user name is checked for validity.

The form of the logon message is:

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

The length of this message is 54 bytes.

The form of the reply to the logon message is:

```
STRUCT logon^reply;
  BEGIN
    INT replycode;            !  [0] 0 = allow logon.
                              !      1 = disallow logon.
    STRING                    !
        replytext [ 0:131 ]; !  [1] optional message to be printed.
  END;
```

The length of this message is (2 + reply text length) bytes. Note that the length of the reply text is implied in the reply count used when making a reply. If reply count = 2, no text is displayed.

LOGOFF MESSAGE: This message is sent to the $CMON process when a LOGOFF command is entered.

The form of the logoff message is:

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

The length of this message is 54 bytes.

The form of the reply to the logoff message is:

```
STRUCT logoff^reply;
  BEGIN
    INT replycode;              !  [0] 0 or 1.
    STRING                      !
        replytext [ 0:131 ]; !  [1] optional message to be printed.
  END;
```

The length of this message is (2 + reply text length) bytes. Note that the length of the reply text is implied in the reply count used when making a reply. If reply count = 2, no text is displayed.

PROCESS CREATION MESSAGE: This message is sent to the $CMON process when an implicit or explicit RUN command is entered.

The form of this message is:

```
STRUCT processcreation^msg;
  BEGIN
    INT msgcode,                !  [0] -52.
        userid,                 !  [1] user ID of user logging on.
        cipri,                  !  [2] execution priority if CI.
        ciinfile [ 0:11 ],      !  [3] name of CI's <command file>.
        cioutfile[ 0:11 ],      ! [15] name of CI's <list file>.
        progname [ 0:11 ],      ! [27] expanded program file name.
        priority,               ! [39] <priority> of RUN command if
                                !      supplied; otherwise -1.
        processor;              ! [40] <processor module> of RUN
                                !      command if supplied;
                                !      otherwise, -1.
  END;
```

The length of this message is 82 bytes.

The $CMON process may reply in either of two ways. The first causes a process creation to be attempted. This form of reply is:

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

The values returned in this reply are those used for the process creation attempt. Any process creation errors are seen by the Command Interpreter user (no notification is made to $CMON).

The second form of reply is used to disallow the process creation. This form of reply is:

```
STRUCT processcreation^reply;
  BEGIN
    INT replycode;              !  [0] 1 = disallow process creation.
    STRING                      !
        replytext [ 0:131 ];    !  [1] optional message to be printed.
  END;
```

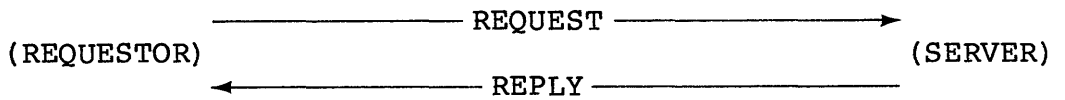
The length of this message is (2 + reply text length) bytes. Note that the length of the reply text is implied in the reply count used when making a reply. If reply count = 2, no text is displayed.

SECTION 12

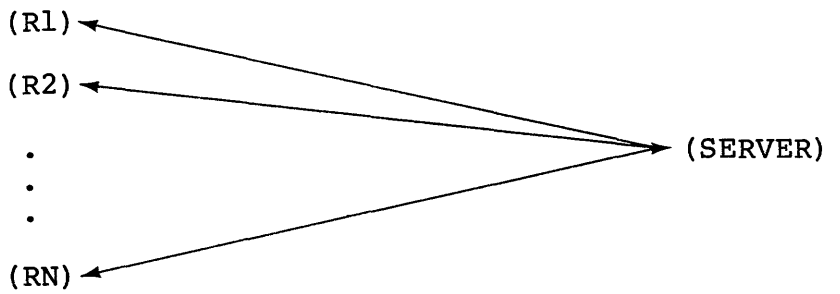
NonStop PROGRAMMING EXAMPLE

This section presents an example of NonStop programming technique. The example program is a NonStop server process.

A "server" process is an application process that accepts a request from a "requestor" process, fulfills the request, then returns a reply (usually consisting of a data message or an error indication) to the requestor:



Typically, a server process can accept requests from multiple requestor processes:



Server processes are used in instances where

- it is desired to modularize the application by function.
- several application programs need to execute complex, but similar functions. In this case, a server process can perform these functions on behalf of the requestor processes. This eliminates the need for each program to contain large amounts of similar coding.
- it is desired that each terminal in an application be controlled by a separate process, and several terminals (i.e., processes) must access the same set of disc files, but it is undesirable for each process to have the disc files open. In this case, a server

process performs all the disc operations, and therefore is the only process with the disc files open. The terminal requestor processes perform disc operations by sending requests to the server process using interprocess communication methods. This method of disc file handling can also be used to reduce or eliminate the need for file locking.

- a custom interface to a non-standard i/o device is needed. In this case, the server process translates normal file system WRITE or WRITEREAD requests into the specific file system requests needed to control the device. An example of this is a server process that interfaces, via ENVOY protocols, to a data communications line; requests are made to the server as though communicating with a conversational mode terminal (i.e., via WRITEREAD), and the server translates the request into the WRITEs, READs, and CONTROLs needed to control the line.

The example presented here is a server process that accesses a data base (i.e., disc file) on behalf of several requestor processes (i.e., terminals). As such, the example program is a culmination of the programming techniques discussed in section 2.9, "Interprocess Communication", and section 5.3, "Using the Checkpointing Facility".

THE NonStop EXAMPLE PROGRAM

The example program (figure 12-1) is called "serveobj", and its source program is called "servesrc". It executes as a process pair in two processor modules. One process of the pair is the primary process; it performs the requested operations. The other process of the pair is the backup process; it monitors the operational state of the primary. If the primary server process becomes inoperable, the backup process takes over and performs the server function.

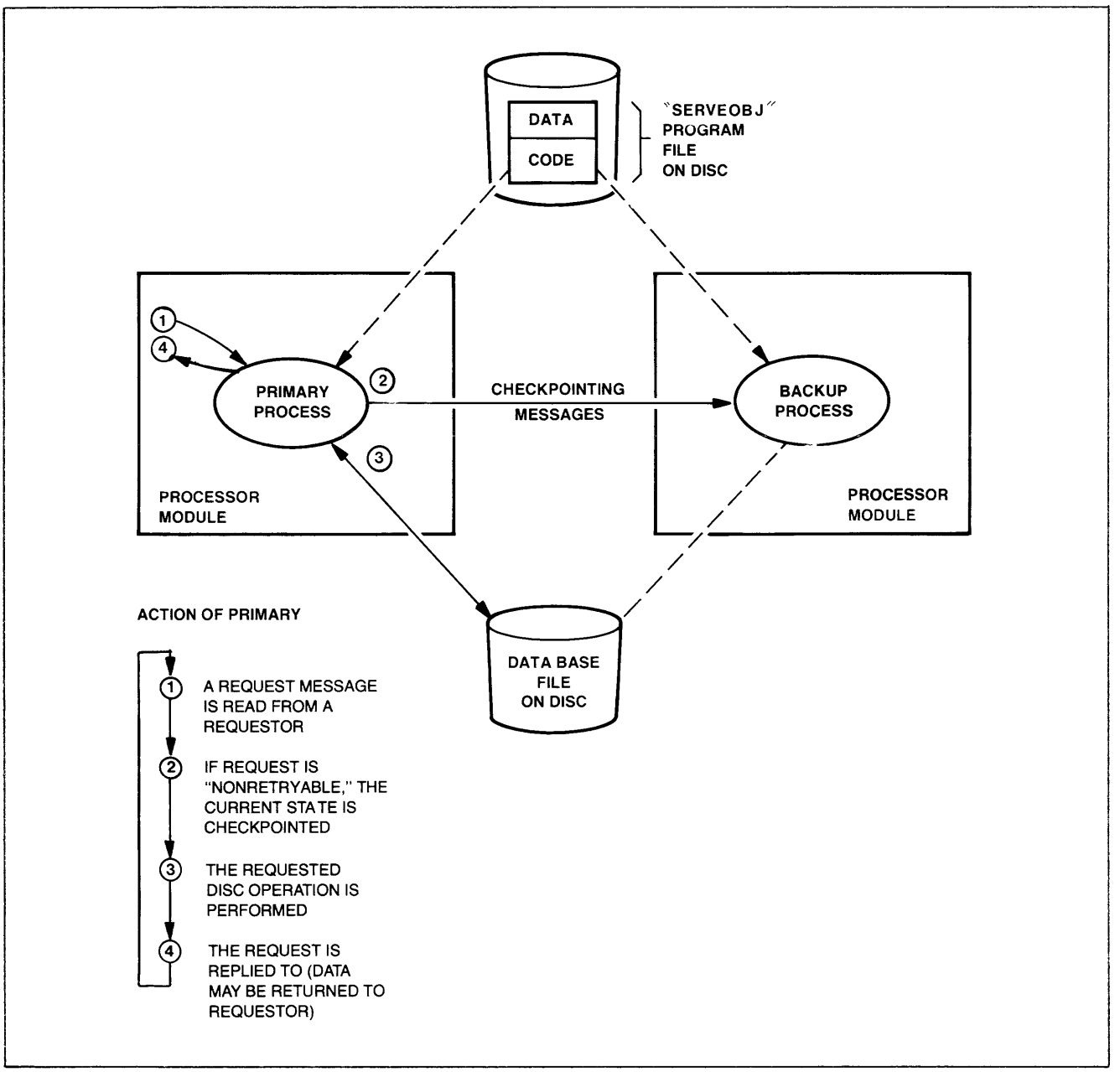


Figure 12-1. "Serveobj" Program

When the "servesrc" program is initially run (e.g., by a Command Interpreter RUN command), it is given the process name "$SERVE" (requestor processes access the server by this name). The first process created assumes the role of the primary process. The primary process, after successfully opening its files, creates the backup process, then opens the backup's files. The backup process, as soon as it determines that it is the backup, begins monitoring the primary process.

The function of the primary process is to read a request message, perform the action indicated by the request, then return the outcome of the request - an error code and/or data - in a reply message to the requestor. The requests that the server processes are

- insert: Insert the supplied record into the data base.
- delete: Delete the supplied record from the data base.
- query: Return the record from the data base as indicated by the supplied key value.
- next: Return the next record from the data base as indicated by the supplied key value.

Example Program Structure

The example program consists of the following procedures:

- "serve"

  This is the main procedure. It is where the primary/backup determination is made. If the process is the primary, the startup message is read, the data base file is opened, the backup is created, and the main processing loop is called. If the process is the backup, the CHECKMONITOR procedure is called.

- "execute"

  This is the execution loop of the server process. It waits on the $RECEIVE file for incoming messages. If a user request message is received, the "process^user^request" procedure is called. If a system message is received, the "analyze^system^message" procedure is called.

- "process^user^request"

  This procedure interprets the user request, checks for duplicate requests, checkpoints in some cases, calls the appropriate "primitive" to process the user request, and saves the outcome of the operation for the requestor. The user request primitives are

  - "query^request": get the record from the data base as indicated by the supplied key value.

  - "insert^request": insert the supplied record into the data base.

  - "delete^request": delete the record from the data base as indicated by the supplied key value.

  - "next^record^request": get the next record from the data base as indicated by the supplied key value.

- "analyze^system^message"

  This procedure interprets system messages and takes appropriate action. This involves creating a backup process in some cases, and adding and deleting requestors from the "requestor process directory".

  The "requestor process directory" is a table of all processes (and their backups) currently accessing the server. This is maintained so that the server can ensure that the proper reply is made to a requestor in the event of a requestor or server failure (see "Request Integrity"). There are four "primitives" used to perform directory operations. They are:

  - "lookuppid": look up a requestor in the directory (called when a user request message is received).

  - "addpid": add a requestor to the directory (called when an OPEN system message is received).

  - "delpid": delete a requestor from the directory (called when a CLOSE system message is received).

  - "delallpids": delete all requestors from the directory that are associated with a given cpu (called if a cpu failure occurs).

- "analyze^checkpoint^status"

  This procedure is called when a nonzero return is made from the CHECKPOINT procedure. This usually occurs only when the backup takes over. This procedure analyzes the reason for the takeover and takes appropriate action.

- "create^backup"

  This procedure performs the backup process creation function. It is called at the start of primary execution, called when the primary detects a failure of the backup and the backup processor module is operable, and called on a takeover by the backup when the primary process failed because of an ABEND condition (e.g., a trap). Following a successful creation of the backup process, the files are opened for the backup, and the current state of the primary process is checkpointed.

- "open^primarys^files" and "open^backups^files"

  These two procedures perform the file open functions for the primary process and the backup process, respectively.

- "read^start^up^message"

  This procedure is called at the beginning of the primary process's execution to open the $RECEIVE file, read the startup message, and save the file name of the data base disc file.

Request Integrity

For the purpose of preventing erroneous results being returned to a requestor if a failure of a requestor or the server occurs, requests are classified by the server as being either retryable or nonretryable. The "retryable" requests are those which do not alter the data base, and therefore can be reexecuted indefinitely with the same results. The retryable requests are "query" and "next". The "nonretryable" requests, conversely, alter the data base and would return different results if reexecuted (e.g., the first insert of a given record is successful, but the second insert of the same record results in a "record already exists" error). The nonretryable requests are "insert" and "delete".

Each request message contains a sync ID. The sync ID is used by the server process to detect duplicate requests for nonretryable operations (i.e., "insert" or "delete") from a given requestor process (duplicate requests are caused by a failure of a requestor process). The value of the sync ID is incremented by the requestor with each request. When a new request is received, and the request is nonretryable, the server saves the value of the sync ID (these are saved for each given requestor). If the sync ID in a request does not match the saved sync ID for the requestor, then the request is a new request. In this case, the requested operation is performed, the results are returned to the requestor, and the error code which was returned to the requestor (to indicate the outcome of the operation) is saved by the server process for the requestor. (Note that, because the server only saves the sync ID's associated with nonretryable requests, the sync ID associated with a retryable request will always indicate a new request. Therefore, duplicate retryable requests will be reexecuted by the server). If the sync ID in a request message matches the saved sync ID, then the request is a duplicate request for a nonretryable operation. The server does not reexecute the operation. Rather, it returns the completion status that it saved for that requestor.

Checkpoints

There are three types of checkpoints in the example program:

- initial checkpoint
- request checkpoint
- process requestor directory checkpoint

INITIAL CHECKPOINT: The initial checkpoint is made to the backup process, in the "create^backup" procedure", following the successful creation of the backup and the opening of its files. The checkpoint includes the entire data area from 'G'[5] through the top-of-stack location, and includes the "sync block" for the data base file (the variable "backup^cpu" is not checkpointed). At this point, the backup process's data area is an exact copy of the primary process's data area.

REQUEST CHECKPOINT (figure 12-2): Each time through its main processing loop in the "process^user^request" procedure, if the current request is nonretryable, the primary checkpoints the outcome of the preceding nonretryable request and the state of the current request to the backup. This is accomplished via a call to the CHECKPOINT procedure. The purpose of the checkpoint is to keep the backup informed as to the current state of the primary and to define a restart point for the backup in the event that the primary fails.

In the example program, the goal was to keep the number of checkpoints and the amount of data checkpointed to a minimum. It is important to note that the checkpoint is made only if the current request is nonretryable, and that this one checkpoint per processing loop is ample for all failure recovery. The data checkpointed is:

- the data stack. This is kept small by the use of global data buffers.
- the data base file sync block
- the sync ID, by requestor, of the current request
- the data record
- the error return value, by requestor, of the preceding nonretryable request

Any time a failure of the server primary occurs, the backup takes over from the latest checkpoint (which is for the latest nonretryable operation) and reexecutes the latest nonretryable operation. (Note that this generates the result value for the latest nonretryable operation. The backup now has the correct values for any nonretryable operation which had been performed by the primary.) If the failure occurs between the call to READUPDATE and the call to CHECKPOINT, the backup is reexecuting an operation already replied to. If the failure occurs between the call to CHECKPOINT and the call to REPLY, the backup completes the operation associated with the current request. In either case, the use of the "sync block" by the file system ensures that the request is not duplicated. Note also that the backup executes the call to REPLY at the end of the processing loop. This call is rejected, because there is no outstanding request on the backup side at this time (the rejection is ignored).

If the primary server process was in the midst of processing a request when a failure occurred, the backup server process receives the same request when it takes over. If the primary had not reached the checkpoint for the current operation, then the backup has the value of the preceding nonretryable sync ID. The backup sees the request as a

new request and processes it. If the primary was processing a nonretryable request and had reached the checkpoint for the current operation, then the current sync ID was checkpointed (and, in fact, the backup completed the operation on its takeover). The backup sees this request as a duplicate request and returns the saved error code for the completed operation.

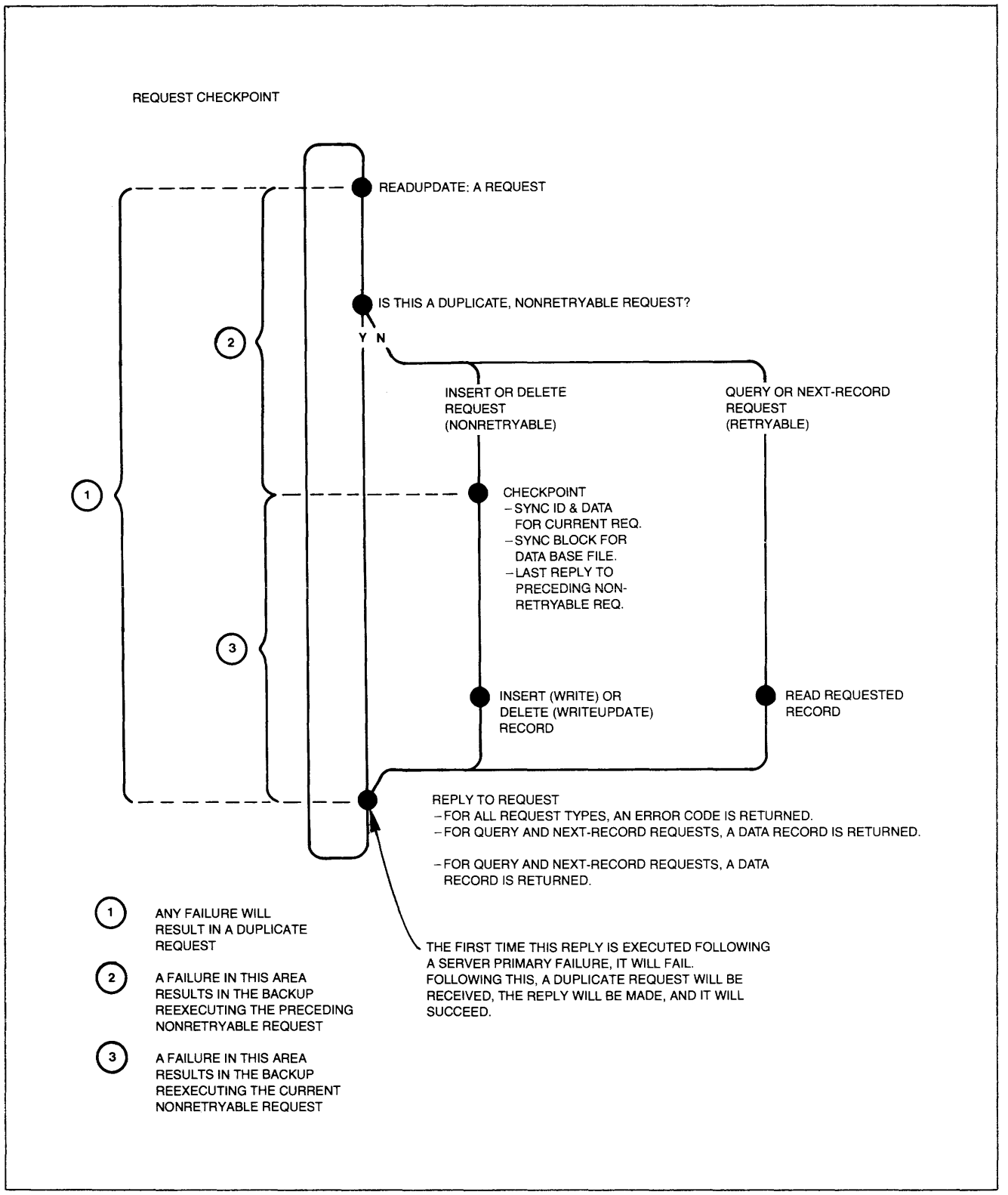


Figure 12-2. Request Checkpoint

PROCESS REQUESTOR DIRECTORY CHECKPOINT: This checkpoint is made, following the processing of a system message, in the "analyze^system^message" procedure. It checkpoints the entire requestor process directory and the entire sync count table. This point was chosen for the directory checkpoint because directory changes may occur when system messages are received. (Note that a directory change is also made in the "analyze^checkpoint^status" procedure if the takeover was because of a processor module failure. However, a checkpoint at this point would be useless.)

```
PAGE 1    $BOOKS1.M096B01.SERVESRC [1]

TAL - T9200D03 - (01JUL81)                          Source language: TAL - Target machine: Tandem NonStop System
Date - Time :  2/23/82 - 13:43:07          Default options: On (LIST,CODE,MAP,WARN,LMAP) - Off (ICODE,INNERLIST)

    1.    000000 0   0  ! NonStop Programming Example
    2.    000000 0   0  ! for GUARDIAN Programming Manual
    3.    000000 0   0  ! P/N 82096 (TNS and TNS/II)
    4.    000000 0   0
    5.    000000 0   0  ! Server Program
    6.    000000 0   0
    7.    000000 0   0  ?NOLIST
   10.    000000 0   0
   12.    000000 0   0
   13.    000000 0   0  ?LMAP*
   14.    000000 0   0
   15.    000000 0   0  ! This program is run by the following RUN command:
   16.    000000 0   0  !
   17.    000000 0   0  !    RUN serveobj / IN <data base file> , NAME $SERVE /
   18.    000000 0   0  !    --- -------- - -- ----------------- - ---- ------ -
   19.    000000 0   0  !
   20.    000000 0   0  !    where
   21.    000000 0   0  !
   22.    000000 0   0  !       <data base file> is the file name of the disc file to ba accessed.
   23.    000000 0   0  !
   24.    000000 0   0  !       $SERVE is the name used by requestors to access this process.
   25.    000000 0   0  !
   26.    000000 0   0  ! process NonStop state.
   27.    000000 0   0  INT backup^cpu,        ! backup cpu number.              ** NOT CHECKPOINTED.
   28.    000000 0   0      backup^pid[0:3],  ! process id of backup process.   ** NOT CHECKPOINTED.
   29.    000000 0   0      stop^count := 0,  ! to detect looping backup delete-creates
   30.    000000 0   0      backup^up :=  0,  ! true if backup is running.
   31.    000000 0   0     .stack^base;       ! beginning of data stack for checkpointing.
   32.    000000 0   0
   33.    000000 0   0  ! files
   34.    000000 0   0  INT recv^fnum,         ! $RECEIVE file number.
   35.    000000 0   0      db^fnum,          ! data base file number.
   36.    000000 0   0      db^fname[0:11];   ! data base file name.
   37.    000000 0   0
   38.    000000 0   0  LITERAL open^msgs        = %40000,
   39.    000000 0   0          protected       = %40,
   40.    000000 0   0          no^wait         = 1,
   41.    000000 0   0          recv^flags      = open^msgs + no^wait,
   42.    000000 0   0          recv^sync^depth = 1,
   43.    000000 0   0          db^flags        = protected,
   44.    000000 0   0          db^sync^depth   = 1;
   45.    000000 0   0
   46.    000000 0   0  ! data base format.
   47.    000000 0   0  LITERAL db^rec^len     = 256,
   48.    000000 0   0          db^rec^key^off =   0,
   49.    000000 0   0          db^rec^key^len =  24;
   50.    000000 0   0
   51.    000000 0   0  ! request message format.
   52.    000000 0   0  !
   53.    000000 0   0  !   word
   54.    000000 0   0  !   [0]     [1]              [2:message^size-3]
   55.    000000 0   0  !   [ sync ][ request type ][ -------------------- record -------------------- ]
   56.    000000 0   0  !
```

```
000000 0  0  !   sync = sync id ( incremented by requester on each request ).
000000 0  0  !          used to detect duplicate nonretryable requests.
000000 0  0  !
000000 0  0  !   request type.
000000 0  0  !
000000 0  0  !      0 = insert request       (nonretryable)
000000 0  0  !      1 = delete request.      (nonretryable)
000000 0  0  !      2 = query request.       (retryable)
000000 0  0  !      3 = next record request. (retryable)
000000 0  0  !
000000 0  0  !   record.
000000 0  0  !
000000 0  0  !      on request = record to be inserted, deleted, or obtained.
000000 0  0  !      on return for query or next = record from data base.
000000 0  0
000000 0  0  LITERAL
000000 0  0       sync              = 0,
000000 0  0       request^type      = 1,
000000 0  0       record            = 2,
000000 0  0       message^len       = db^rec^len + 4,
000000 0  0       message^size      = message^len / 2;
000000 0  0
000000 0  0  ! global buffers.
000000 0  0  INT .recv^buf[ 0 : message^size ],   ! receive buffer.
000203 0  0       recv^cnt,                       ! receive count.
000203 0  0      .buf     [ 0 : db^rec^len / 2 ]; ! scratch buffer.
000404 0  0
000404 0  0  LITERAL
000404 0  0       max^reqstrs = 16; ! maximum number of requestors allowed.
000404 0  0
000404 0  0  INT
000404 0  0       ! directory entry no. of previous requestor of nonretryable operation.
000404 0  0       old^requestor,
000404 0  0       ! sync id for latest requestor of nonretryable operation.
000404 0  0      .sync^count  [ 1 : max^reqstrs ] := max^reqstrs * [ 0 ],
000424 0  0       ! reply error code for each requestor.
000424 0  0      .reply^error [ 1 : max^reqstrs ],
000444 0  0       ! reply for current request.
000444 0  0      .reply^buf   [ 0 : message^size - 1 ];
000646 0  0
000646 0  0  ! requestor process directory.
000646 0  0  INT .pids[5:max^reqstrs * 5 + 5] := (max^reqstrs * 5) * [ 0 ];
000767 0  0
000767 0  0  !   [0]        [3] [4]
000767 0  0  !
000767 0  0  !   ┌──────────┬───┬───┐  entry #1
000767 0  0  !   │          │   │   │
000767 0  0  !   ├──────────┼───┼───┤  entry #2
000767 0  0  !   │          │   │   │
000767 0  0  !   ├──────────┼───┼───┤  entry #3
000767 0  0  !   │          │   │   │     .
000767 0  0  !   ~          ~   ~   ~     .
000767 0  0  !                           entry #max^reqstrs.
000767 0  0  !
000767 0  0  !
000767 0  0  !   entry^no[0:2] = <process name> or <creation time stamp>
000767 0  0  !   entry^no[3]   = <cpu,pin> of primary process
```

```
114.    000767 0  0  !   entry^no[4]    = <cpu,pin> of backup process, if any, or zero
115.    000767 0  0
116.    000767 0  0  ?NOLIST
```

PAGE 4 $BOOKS1.M096B01.SERVESRC [1] SERVE: read^start^up^message Procedure 2/23/82 13:

```
177.  000000 0  0  ! this procedure opens the $RECEIVE file for the primary process and reads
178.  000000 0  0  ! the start-up message.  Note that the receipt of the start-up message
179.  000000 0  0  ! involves reading three interprocess messages:
180.  000000 0  0  !
181.  000000 0  0  !  1.  OPEN "system" message.
182.  000000 0  0  !  2.  CI start-up message (not a "system" message)
183.  000000 0  0  !  3.  CLOSE "system" message.
184.  000000 0  0
185.  000000 0  0  PROC read^start^up^message;
186.  000000 1  0
187.  000000 1  0    BEGIN
188.  000000 1  1      INT .recv^buf[0:33];
189.  000000 1  1
190.  000000 1  1      ! open $RECEIVE.
191.  000000 1  1      recv^buf ':=' ["$RECEIVE", 8 * [ "  " ]];
192.  000011 1  1      CALL OPEN ( recv^buf, recv^fnum, recv^flags, recv^sync^depth );
193.  000023 1  1      IF <> THEN CALL ABEND;
194.  000027 1  1
195.  000027 1  1      ! read open message.
196.  000027 1  1      CALL READ ( recv^fnum, recv^buf, 14 );
197.  000037 1  1      CALL AWAITIO ( recv^fnum,,,, 3000D );
198.  000050 1  1      IF <= OR recv^buf <> -30 THEN CALL ABEND;
199.  000057 1  1
200.  000057 1  1      ! read start^up message.
201.  000057 1  1      CALL READ ( recv^fnum, recv^buf, 66 );
202.  000067 1  1      CALL AWAITIO ( recv^fnum,,,, 3000D );
203.  000100 1  1      IF <> OR recv^buf <> -1 THEN CALL ABEND;
204.  000107 1  1
205.  000107 1  1      ! save data^base file name.
206.  000107 1  1      db^fname ':=' recv^buf[9] FOR 12;
207.  000114 1  1
208.  000114 1  1      ! read close message.
209.  000114 1  1      CALL READ ( recv^fnum, recv^buf, 14 );
210.  000124 1  1      CALL AWAITIO ( recv^fnum,,,, 3000D );
211.  000135 1  1      IF <= OR recv^buf <> -31 THEN CALL ABEND;
212.  000144 1  1    END; ! read^start^up^message.

RECV^BUF                        Variable        INT        L+001    Indirect

00000  070402 024700 002042 170401 000025 020137 000200 100014     00010 026047 170401 070010 005100 004001 100001 024733 002
00020  100360 024700 027000 012003 000002 024711 027000 040010     00030 170401 100016 024722 002003 100034 024700 027000 070
00040  024700 002003 100000 005013 004270 100021 024722 027000     00050 016003 140401 001742 012003 000002 024711 027000 040
00060  170401 100102 024722 002003 100034 024700 027000 070010     00070 024700 002003 100000 005013 004270 100021 024722 027
00100  015003 140401 001777 012003 000002 024711 027000 070012     00110 103011 173401 100014 026007 040010 170401 100016 024
00120  002003 100034 024700 027000 070010 024700 002003 100000     00130 005013 004270 100021 024722 027000 016003 140401 001
00140  012003 000002 024711 027000 125003 000141 022122 042503     00150 042511 053105 020040 020040 020040 020040 020040 020
00160  020040 020040
```

```
PAGE 5    $BOOKS1.M096B01.SERVESRC [1] SERVE: File Open Procedures                2/23/82   13:

  214.    000000 0  0  ! this procedure opens the data base file for the primary process.
  215.    000000 0  0
  216.    000000 0  0  PROC open^primarys^files;
  217.    000000 1  0
  218.    000000 1  0    BEGIN
  219.    000000 1  1
  220.    000000 1  1      ! open data base file.
  221.    000000 1  1      CALL OPEN ( db^fname, db^fnum, db^flags, db^sync^depth );
  222.    000011 1  1      IF <> THEN CALL ABEND;
  223.    000015 1  1    END; ! open^primarys^files.


00000  070012 070011 100040 100001 024733 002004 100360 024700     00010 027000 012003 000002 024711 027000 125003

  224.    000000 0  0  ! this procedure opens the $RECEIVE and data base files for the backup
  225.    000000 0  0  ! process.
  226.    000000 0  0
  227.    000000 0  0  PROC open^backups^files;
  228.    000000 1  0
  229.    000000 1  0    BEGIN
  230.    000000 1  1      INT .buf[0:11],
  231.    000000 1  1          back^error;
  232.    000000 1  1
  233.    000000 1  1      ! open $RECEIVE.
  234.    000000 1  1      buf ':=' ["$RECEIVE", 8 * [ "  " ]];
  235.    000011 1  1      CALL CHECKOPEN ( buf, recv^fnum, recv^flags, recv^sync^depth,,,
  236.    000011 1  1                       back^error );
  237.    000024 1  1      IF <> THEN CALL STOP ( backup^pid );
  238.    000031 1  1
  239.    000031 1  1      ! open data base file.
  240.    000031 1  1      CALL CHECKOPEN ( db^fname, db^fnum, db^flags, db^sync^depth,,,
  241.    000031 1  1                       back^error );
  242.    000043 1  1      IF <> THEN CALL STOP ( backup^pid );
  243.    000050 1  1    END; ! open^backups^files.

BACK^ERROR                      Variable        INT       L+002    Direct
BUF                             Variable        INT       L+001    Indirect


00000  070403 024700 002015 170401 000025 020043 000200 100014     00010 026047 170401 040010 005100 004001 100001 000002 070
00020  024766 100171 024700 027000 012004 070001 100001 024711     00030 027000 070012 040011 100040 100001 000002 070402 024
00040  100171 024700 027000 012004 070001 100001 024711 027000     00050 125003 000045 022122 042503 042511 053105 020040 020
00060  020040 020040 020040 020040 020040 020040
```

```
PAGE 6    $BOOKS1.M096B01.SERVESRC [1] SERVE: create^backup Procedure                    2/23/82   13:

  245.    000000 0  0  ! this procedure creates the backup process.  The steps involved are:
  246.    000000 0  0  !
  247.    000000 0  0  ! - create backup.
  248.    000000 0  0  ! - open its files.
  249.    000000 0  0  ! - checkpoint current state.
  250.    000000 0  0
  251.    000000 0  0  PROC create^backup ( backup^cpu );
  252.    000000 1  0     INT backup^cpu;
  253.    000000 1  0
  254.    000000 1  0    BEGIN
  255.    000000 1  1      INT .pfile[0:11],
  256.    000000 1  1          pname[0:3],
  257.    000000 1  1          error,
  258.    000000 1  1          status,
  259.    000000 1  1         .globals := 5; ! base for initial checkpoint = 'G'[5].  "backup^cpu"
  260.    000000 1  1                        ! and "backup^pid" not checkpointed.
  261.    000000 1  1
  262.    000000 1  1      ! check for looping creates.
  263.    000000 1  1      IF stop^count > 5 THEN CALL DEBUG;
  264.    000012 1  1
  265.    000012 1  1      ! get program's file name.
  266.    000012 1  1      CALL PROGRAMFILENAME ( pfile );
  267.    000015 1  1      ! get process's name.
  268.    000015 1  1      CALL GETCRTPID ( MYPID, pname );
  269.    000021 1  1      ! create the backup process.
  270.    000021 1  1      CALL NEWPROCESS ( pfile,,, backup^cpu, backup^pid, error, pname );
  271.    000033 1  1      IF backup^pid THEN ! it was created.
  272.    000035 1  1        BEGIN
  273.    000035 1  2          backup^up := 1;
  274.    000037 1  2          CALL open^backups^files;
  275.    000040 1  2
  276.    000040 1  2          ! checkpoint global area through top-of-stack and sync block.
  277.    000040 1  2          IF ( status := CHECKPOINT ( globals, ,db^fnum ) ) THEN          ! *** CHECKPOINT *** !
  278.    000053 1  2            CALL analyze^checkpoint^status ( status );
  279.    000056 1  2        END;
  280.    000056 1  1    END; ! of create^backup

BACKUP^CPU                      Variable        INT       L-003    Direct
ERROR                           Variable        INT       L+006    Direct
GLOBALS                         Variable        INT       L+010    Indirect
PFILE                           Variable        INT       L+001    Indirect
PNAME                           Variable        INT       L+002    Direct
STATUS                          Variable        INT       L+007    Direct

00000  070411 024700 002006 100005 024700 002014 040005 001005     00010 016001 027000 170401 024700 027000 027000 070402 024
00020  027000 170401 000002 040703 070001 070406 024755 070402     00030 100117 024711 027000 040001 014421 100001 044006 027
00040  170410 100000 040011 024722 002030 005005 100000 024711     00050 027000 034407 014403 040407 024700 027000 125004
```

```
  282.    000000 0  0  ! this procedure is used to analyze and take appropriate action for a
  283.    000000 0  0  ! non-zero return from the CHECKPOINT procedure.
  284.    000000 0  0
  285.    000000 0  0  PROC analyze^checkpoint^status ( status );
  286.    000000 1  0     INT status; ! return value of CHECKPOINT.
  287.    000000 1  0
  288.    000000 1  0    BEGIN
  289.    000000 1  1
  290.    000000 1  1      IF backup^up THEN ! analyze^it.
  291.    000002 1  1        CASE status.<0:7> OF
  292.    000005 1  1          BEGIN
  293.    000005 1  2  ! 0 !     ;     ! good checkpoint.
  294.    000005 1  2
  295.    000005 1  2  ! 1 !     BEGIN ! checkpoint failure.
  296.    000005 1  3              ! find out if backup is still running.
  297.    000005 1  3              CALL GETCRTPID ( backup^pid[3], backup^pid );
  298.    000011 1  3              IF = THEN ! backup still running.
  299.    000012 1  3                BEGIN
  300.    000012 1  4                  ! stop the backup.
  301.    000012 1  4                  CALL STOP ( backup^pid );
  302.    000016 1  4                  backup^up := 0;
  303.    000020 1  4                END;
  304.    000020 1  3            END;  ! 1.
  305.    000021 1  2
  306.    000021 1  2  ! 2 !     BEGIN ! takeover from primary.
  307.    000021 1  3              CASE status.<8:15> OF
  308.    000024 1  3                BEGIN
  309.    000024 1  4
  310.    000024 1  4  ! 2-0 !         BEGIN ! primary stopped, so stop myself.
  311.    000024 1  5                    CALL STOP;
  312.    000027 1  5                  END;
  313.    000030 1  4
  314.    000030 1  4  ! 2-1 !         BEGIN ! primary abended, so create a backup for me.
  315.    000030 1  5                    backup^up := 0;
  316.    000032 1  5                    stop^count := stop^count + 1;
  317.    000034 1  5                    CALL create^backup ( backup^cpu );
  318.    000037 1  5                  END;
  319.    000040 1  4
  320.    000040 1  4  ! 2-2 !         BEGIN ! primary cpu down, note it.
  321.    000040 1  5                    backup^up := 0;
  322.    000042 1  5                    CALL delallpids ( backup^cpu );
  323.    000045 1  5                  END;
  324.    000046 1  4
  325.    000046 1  4  ! 2-3 !         ! primary called CHECKSWITCH.
  326.    000046 1  4                  ;
  327.    000046 1  4                END; ! case of status.<8:15>.
  328.    000053 1  3            END;  ! 2.
  329.    000054 1  2
  330.    000054 1  2  ! 3 !     BEGIN ! bad parameter to CHECKPOINT
  331.    000054 1  3              CALL DEBUG;
  332.    000055 1  3            END;  ! 3.
  333.    000056 1  2
  334.    000056 1  2          END; ! case of status.<0:7>.
  335.    000063 1  1    END; ! analyze^checkpoint^status.

STATUS                          Variable        INT        L-003    Direct
```

```
00000  040006 014461 040703 030110 010451 040004 070001 024711     00010 027000 015006 070001 100001 024711 027000 100000 044
00020  010442 040703 006377 010422 000002 024711 027000 010423     00030 100000 044006 100001 074005 040000 024700 027000 010
00040  100000 044006 040000 024700 027000 010405 000030 177755     00050 177760 177767 000001 010407 027000 010405 000030 000
00060  177725 177740 177772 125004
```

```
  337.    000000 0  0  ! This procedure searches the requestor process directory by a process id
  338.    000000 0  0  ! for an entry number.
  339.    000000 0  0  !
  340.    000000 0  0  !   return values:
  341.    000000 0  0  !     0 = pid not in directory.
  342.    000000 0  0  !    >0 = entry no of pid in directory.
  343.    000000 0  0
  344.    000000 0  0  INT PROC lookuppid(pid);
  345.    000000 1  0     INT .pid;
  346.    000000 1  0
  347.    000000 1  0    BEGIN
  348.    000000 1  1      INT entry^no := 0, ! entry^no in local pid directory.
  349.    000000 1  1          comp^len;      ! compare length for pid matching.
  350.    000000 1  1
  351.    000000 1  1      comp^len := IF pid.<0:7> = "$" THEN ! process name ! 3 ELSE 4;
  352.    000013 1  1      WHILE (entry^no := entry^no + 1) <= max^reqstrs DO
  353.    000020 1  1        IF pid = pids[entry^no * 5] FOR comp^len THEN ! found it.
  354.    000031 1  1           RETURN entry^no;
  355.    000034 1  1
  356.    000034 1  1      RETURN 0; ! not found.
  357.    000036 1  1    END; ! lookuppid.

COMP^LEN                        Variable        INT        L+002    Direct
ENTRY^NO                        Variable        INT        L+001    Direct
PID                             Variable        INT        L-003    Indirect

00000  100000 024700 002001 140703 030110 001044 015002 100003     00010 010401 100004 044402 040401 104001 034401 001020 011
00020  170703 040401 100005 000212 000117 173035 040402 026207     00030 015002 040401 125004 010757 100000 125004
```

```
PAGE 10   $BOOKS1.M096B01.SERVESRC [1] SERVE: addpid Procedure                2/23/82   13:

  359.    000000 0  0  ! This procedure adds a process id to the requestor process directory.
  360.    000000 0  0  !
  361.    000000 0  0  !   return values.
  362.    000000 0  0  !     0 = directory full, "pid" not added to directory.
  363.    000000 0  0  !    >0 = "pid" added, entry no of "pid" in directory.
  364.    000000 0  0
  365.    000000 0  0  INT PROC addpid(pid);
  366.    000000 1  0     INT .pid;
  367.    000000 1  0
  368.    000000 1  0    BEGIN
  369.    000000 1  1      INT entry^no,                  ! entry^no in local pid directory.
  370.    000000 1  1          zero[0:3] := [ 4 * [ 0 ]]; ! for lookup of empty directory slot.
  371.    000004 1  1
  372.    000004 1  1      IF (entry^no := lookuppid(pid)) THEN  ! already in directory.
  373.    000017 1  1        BEGIN
  374.    000017 1  2          ! check for duplicate open.
  375.    000017 1  2          IF pids[ entry^no * 5 + 3 ] <> pid[ 3 ] AND
  376.    000017 1  2             pids[ entry^no * 5 + 4 ] <> pid[ 3 ] THEN ! first open by
  377.    000042 1  2             pids[ entry^no * 5 + 4 ] := pid[ 3 ];     ! backup.
  378.    000044 1  2        END
  379.    000044 1  1      ELSE ! not in directory.  First open by "pid"
  380.    000045 1  1        BEGIN
  381.    000045 1  2          IF (entry^no := lookuppid(zero)) THEN ! look for empty slot.
  382.    000052 1  2            BEGIN
  383.    000052 1  3              pids[entry^no * 5] ':=' pid FOR 4;
  384.    000062 1  3              sync^count[entry^no] := -1; ! initialize requestor id^count.
  385.    000065 1  3            END;
  386.    000065 1  2        END;
  387.    000065 1  1      RETURN entry^no;   ! this returns zero if no room in directory.
  388.    000067 1  1    END; ! addpid.

ENTRY^NO                        Variable        INT        L+001    Direct
PID                             Variable        INT        L-003    Indirect
ZERO                            Variable        INT        L+002    Direct


00000  000000 000000 000000 000000 002005 070402 000025 003771     00010 100004 026047 170703 024700 027000 034401 014426 040
00020  100005 000212 104003 000117 143035 102003 142703 000215     00030 012013 040401 100005 000212 104004 000115 141035 142
00040  000215 012002 142703 145035 010420 070402 024700 027000     00050 034401 014413 040401 100005 000212 000117 173035 170
00060  100004 026007 032401 100777 146032 040401 125004
```

```
PAGE 11   $BOOKS1.M096B01.SERVESRC [1] SERVE: delpid Procedure                    2/23/82   13:

  390.    000000 0  0  ! This procedure deletes a process id from the requestor process
  391.    000000 0  0  ! directory.
  392.    000000 0  0
  393.    000000 0  0  PROC delpid(pid);
  394.    000000 1  0    INT .pid; ! "pid" to be deleted.
  395.    000000 1  0
  396.    000000 1  0    BEGIN
  397.    000000 1  1      INT entry^no; ! entry^no in local pid directory.
  398.    000000 1  1
  399.    000000 1  1      IF (entry^no := lookuppid(pid)) THEN  ! delete it.
  400.    000006 1  1        IF pids[entry^no * 5 + 4] THEN ! was open by process-pair.
  401.    000015 1  1          BEGIN
  402.    000015 1  2            IF pids[entry^no * 5 + 3] = pid[3] THEN ! close by primary.
  403.    000027 1  2              ! replace primary entry^no with backup.
  404.    000027 1  2              pids[entry^no * 5 + 3] := pids[entry^no * 5 + 4];
  405.    000031 1  2            ! clear backup entry^no.
  406.    000031 1  2            pids[entry^no * 5 + 4] := 0;
  407.    000040 1  2          END
  408.    000040 1  1        ELSE ! was open by one process.
  409.    000041 1  1          pids[entry^no * 5] ':=' [0,0,0,0];
  410.    000053 1  1    END; ! delpid.

ENTRY^NO                          Variable        INT        L+001    Direct
PID                               Variable        INT        L-003    Indirect


00000  002001 170703 024700 027000 034401 014445 040401 100005     00010 000212 104004 000117 143035 014424 040401 100005 000
00020  104003 000116 142035 101003 141703 000215 015002 143035     00030 146035 040401 100005 000212 104004 000117 100000 147
00040  010412 040401 100005 000212 000117 173035 000025 020004     00050 000200 100004 026047 125004 000006 000000 000000 000
00060  000000
```

```
  412.    000000 0  0  ! this procedure is called if a cpu failure message is received.  It
  413.    000000 0  0  ! deletes all references in the requestor process directory to the
  414.    000000 0  0  ! failed cpu.  This may cause entire entries to be deleted.
  415.    000000 0  0
  416.    000000 0  0  PROC delallpids(cpu);
  417.    000000 1  0     INT cpu; ! processor module number of pids to be deleted.
  418.    000000 1  0
  419.    000000 1  0    BEGIN
  420.    000000 1  1      INT entry^no := 0,   ! entry^no in local pid directory.
  421.    000000 1  1          temp;
  422.    000000 1  1
  423.    000000 1  1      WHILE (entry^no := entry^no + 1) <= max^reqstrs DO
  424.    000010 1  1        BEGIN ! check each entry^no.
  425.    000010 1  2
  426.    000010 1  2          ! check for match with entry^no's primary cpu.
  427.    000010 1  2          IF pids[ entry^no * 5 + 3 ] AND
  428.    000010 1  2             pids[ entry^no * 5 + 3 ].<0:7> = cpu THEN ! primary down
  429.    000024 1  2            ! delete primary process and maybe the entire entry^no.
  430.    000024 1  2            CALL delpid ( pids [ entry^no * 5 ] )
  431.    000030 1  2          ELSE
  432.    000031 1  2          ! check for match with entry^no's backup cpu.
  433.    000031 1  2          IF pids[ entry^no * 5 + 4 ] AND
  434.    000031 1  2             pids[ entry^no * 5 + 4 ].<0:7> = cpu THEN ! backup down.
  435.    000045 1  2            ! clear the backup entry^no.
  436.    000045 1  2            pids[ entry^no * 5 + 4 ] := 0;
  437.    000047 1  2        END;
  438.    000050 1  1    END; ! delallpids.

CPU                             Variable        INT        L-003    Direct
ENTRY^NO                        Variable        INT        L+001    Direct
TEMP                            Variable        INT        L+002    Direct


00000  100000 024700 002001 040401 104001 034401 001020 011040     00010 040401 100005 000212 104003 000117 143035 014412 143
00020  030110 040703 000215 015005 107775 173035 024700 027000     00030 010416 040401 100005 000212 104004 000117 143035 014
00040  143035 030110 040703 000215 015002 100000 147035 010733     00050 125004
```

```
440.    000000 0  0  ! this procedure analyzes and takes appropriate action for system messages.
441.    000000 0  0
442.    000000 0  0  PROC analyze^system^message ( recv^buf, recv^cnt, pid );
443.    000000 1  0     INT .recv^buf,
444.    000000 1  0         recv^cnt,
445.    000000 1  0        .pid;
446.    000000 1  0
447.    000000 1  0    BEGIN
448.    000000 1  1      INT reply^error := 0,
449.    000000 1  1          status;
450.    000000 1  1
451.    000000 1  1      CASE $ABS ( recv^buf ) OF
452.    000007 1  1        BEGIN
453.    000007 1  2  !  0  ! ;
454.    000007 1  2  !  1  ! ;
455.    000007 1  2
456.    000007 1  2  !  2  ! BEGIN ! cpu down.
457.    000007 1  3           ! delete any references in the requestor process
458.    000007 1  3           ! directory to the cpu that failed.
459.    000007 1  3           CALL delallpids ( recv^buf[1] );
460.    000013 1  3           IF recv^buf[1] = backup^cpu THEN backup^up := 0;
461.    000022 1  3         END;
462.    000023 1  2
463.    000023 1  2  !  3  ! BEGIN ! cpu up.
464.    000023 1  3           IF recv^buf[1] = backup^cpu THEN ! backup cpu came up.
465.    000030 1  3             BEGIN
466.    000030 1  4               ! clear stop count.
467.    000030 1  4               stop^count := 0;
468.    000032 1  4               CALL create^backup ( backup^cpu );
469.    000035 1  4             END;
470.    000035 1  3         END;
471.    000036 1  2
472.    000036 1  2  !  4  ! ;
473.    000036 1  2
474.    000036 1  2  !  5  ! BEGIN ! backup stopped.
475.    000036 1  3           backup^up := 0;
476.    000040 1  3           stop^count := stop^count + 1;
477.    000042 1  3           CALL create^backup ( backup^cpu );
478.    000045 1  3         END;
479.    000046 1  2
480.    000046 1  2  !  6  ! BEGIN ! backup abended.
481.    000046 1  3           backup^up := 0;
482.    000050 1  3           stop^count := stop^count + 1;
483.    000052 1  3           CALL create^backup ( backup^cpu );
484.    000055 1  3         END;
485.    000056 1  2
486.    000056 1  2  ! 7-29! ;;;;;;;;;;;;;;;;;;;;;;;;
487.    000056 1  2
488.    000056 1  2  !  30 ! BEGIN ! OPEN system message.
489.    000056 1  3           ! check for no-wait i/o depth > 1.
490.    000056 1  3           IF recv^buf[ 1 ].<12:15> > 1 THEN
491.    000063 1  3             reply^error := 28  ! return illegal no-wait depth error.
492.    000063 1  3           ELSE
493.    000066 1  3           ! try to add opener to directory.
494.    000066 1  3           IF NOT addpid ( pid ) THEN
495.    000072 1  3             reply^error := 12; ! return file in use error.
496.    000074 1  3         END;
```

```
PAGE 14   $BOOKS1.M096B01.SERVESRC [1] SERVE: analyze^system^message Procedure                    2/23/82   13:

  497.    000075 1  2
  498.    000075 1  2  !  31 ! BEGIN ! CLOSE system message.
  499.    000075 1  3            ! delete closer from requestor process directory.
  500.    000075 1  3            CALL delpid ( pid );
  501.    000100 1  3          END;
  502.    000101 1  2
  503.    000101 1  2          OTHERWISE reply^error := 2;    ! return invalid operation error.
  504.    000103 1  2        END; ! system message case.
  505.    000154 1  1
  506.    000154 1  1      IF ( status := CHECKPOINT ( stack^base,                          ! *** CHECKPOINT *** !
  507.    000154 1  1                                  pids [ 5 ], max^reqstrs * 5 + 5,
  508.    000154 1  1                                  sync^count [ 1 ], max^reqstrs,
  509.    000154 1  1                                  stop^count, 1,
  510.    000154 1  1                                  reply^error [ old^requestor ], 1,
  511.    000154 1  1                                  ,db^fnum) ) THEN
  512.    000204 1  1        CALL analyze^checkpoint^status ( status );
  513.    000207 1  1
  514.    000207 1  1      ! reply to system message.
  515.    000207 1  1      CALL REPLY (,,,, reply^error);
  516.    000214 1  1    END; ! analyze^system^message.

PID                             Variable        INT        L-003    Indirect
RECV^BUF                        Variable        INT        L-005    Indirect
RECV^CNT                        Variable        INT        L-004    Direct
REPLY^ERROR                     Variable        INT        L+001    Direct
STATUS                          Variable        INT        L+002    Direct

00000  100000 024700 002001 140705 013001 000214 010475 103001     00010 143705 024700 027000 103001 143705 040000 000215 015
00020  100000 044006 010531 103001 143705 040000 000215 015005     00030 100000 044005 040000 024700 027000 010516 100000 044
00040  100001 074005 040000 024700 027000 010506 100000 044006     00050 100001 074005 040000 024700 027000 010476 103001 143
00060  006017 001001 016003 100034 044401 010406 170703 024700     00070 027000 015402 100014 044401 010457 170703 024700 027
00100  010453 100002 044401 010450 100037 000205 011002 000100     00110 010401 100040 000030 000041 000040 177672 177705 000
00120  177716 177725 000032 000031 000030 000027 000026 000025     00130 000024 000023 000022 000021 000020 000017 000016 000
00140  000014 000013 000012 000011 000010 000007 000006 000005     00150 000004 177705 177723 177726 170007 103005 173035 100
00160  102001 172032 100020 070005 024755 100001 031031 071401     00170 100001 100000 040011 024744 002020 005007 004375 100
00200  024711 027000 034402 014403 040402 024700 027000 002004     00210 040401 100001 024711 027000 125006
```

```
  518.    000000 0  0  ! this procedure searces the data base for the record associated with
  519.    000000 0  0  ! a key value.
  520.    000000 0  0  !
  521.    000000 0  0  ! return values.
  522.    000000 0  0  !  0 = record found, record in "result".
  523.    000000 0  0  ! >0 = file management error.
  524.    000000 0  0
  525.    000000 0  0  INT PROC query^request ( rec, result, result^len );
  526.    000000 1  0
  527.    000000 1  0     INT .rec,        ! key value.
  528.    000000 1  0         .result,     ! record of key.
  529.    000000 1  0         .result^len; ! length of record of key.
  530.    000000 1  0
  531.    000000 1  0    BEGIN
  532.    000000 1  1      INT error;
  533.    000000 1  1
  534.    000000 1  1      result^len := 0;
  535.    000003 1  1      CALL KEYPOSITION ( db^fnum, rec,,,2 );
  536.    000013 1  1      CALL READUPDATE ( db^fnum, result, db^rec^len, result^len );
  537.    000024 1  1      CALL FILEINFO ( db^fnum, error );
  538.    000033 1  1      RETURN error;
  539.    000035 1  1    END; ! query^request.

ERROR                          Variable        INT        L+001    Direct
REC                            Variable        INT        L-005    Indirect
RESULT                         Variable        INT        L-004    Indirect
RESULT^LEN                     Variable        INT        L-003    Indirect

00000  002001 100000 144703 040011 170705 030001 000002 100002     00010 100031 024755 027000 040011 170704 005001 170703 000
00020  024755 100036 024700 027000 040011 070401 024711 002013     00030 005030 024700 027000 040401 125006
```

```
PAGE 16   $BOOKS1.M096B01.SERVESRC [1] SERVE: insert^request Procedure                    2/23/82   13:

  541.    000000 0  0  ! this procedure adds a record to the data base.
  542.    000000 0  0  !
  543.    000000 0  0  ! return values.
  544.    000000 0  0  !
  545.    000000 0  0  !  0 = record "rec" added.
  546.    000000 0  0  ! >0 = file management error.
  547.    000000 0  0
  548.    000000 0  0  INT PROC insert^request ( rec );
  549.    000000 1  0     INT .rec;     ! record to be added.
  550.    000000 1  0
  551.    000000 1  0    BEGIN
  552.    000000 1  1
  553.    000000 1  1      INT  error;
  554.    000000 1  1
  555.    000000 1  1      ! insert the record.
  556.    000000 1  1      CALL WRITE ( db^fnum, rec, db^rec^len );
  557.    000011 1  1      CALL FILEINFO ( db^fnum, error );
  558.    000020 1  1      RETURN error;
  559.    000022 1  1    END; ! insert^request.

ERROR                          Variable        INT        L+001    Direct
REC                            Variable        INT        L-003    Indirect


00000  002001 040011 170703 005001 024722 002003 100034 024700     00010 027000 040011 070401 024711 002013 005030 024700 027
00020  040401 125004
```

```
  561.    000000 0  0  ! this procedure deletes a record from the data base file.
  562.    000000 0  0  !
  563.    000000 0  0  ! return values.
  564.    000000 0  0  !
  565.    000000 0  0  !   0 = record "rec" deleted.
  566.    000000 0  0  !  >0 = file management error.
  567.    000000 0  0
  568.    000000 0  0  INT PROC delete^request ( rec );
  569.    000000 1  0     INT .rec;     ! key of record to be deleted.
  570.    000000 1  0
  571.    000000 1  0    BEGIN
  572.    000000 1  1      INT  error;
  573.    000000 1  1
  574.    000000 1  1      ! delete the record.
  575.    000000 1  1      CALL KEYPOSITION ( db^fnum, rec );
  576.    000011 1  1      CALL WRITEUPDATE ( db^fnum, rec, 0 );
  577.    000021 1  1      CALL FILEINFO ( db^fnum, error );
  578.    000030 1  1      RETURN error;
  579.    000032 1  1    END; ! delete^request.

ERROR                          Variable        INT        L+001    Direct
REC                            Variable        INT        L-003    Indirect

00000  002001 040011 170703 030001 024711 002003 100030 024700     00010 027000 040011 170703 100000 024722 002003 100034 024
00020  027000 040011 070401 024711 002013 005030 024700 027000     00030 040401 125004
```

```
  581.    000000 0   0  ! this procedure returns the next record in the data base file.
  582.    000000 0   0  !
  583.    000000 0   0  ! calling values.
  584.    000000 0   0  !
  585.    000000 0   0  !   rec = 0, return first record in file.
  586.    000000 0   0  !   rec =  record, return next record in file.
  587.    000000 0   0  !
  588.    000000 0   0  ! return values.
  589.    000000 0   0  !
  590.    000000 0   0  !   0 = first/next record returned in "result".
  591.    000000 0   0  !  >0 = file management error.
  592.    000000 0   0
  593.    000000 0   0  INT PROC next^record^request ( rec, result, result^len );
  594.    000000 1   0    INT .rec,        ! key of record for positioning.
  595.    000000 1   0        .result,     ! next record.
  596.    000000 1   0        .result^len; ! length of next record.
  597.    000000 1   0
  598.    000000 1   0    BEGIN
  599.    000000 1   1
  600.    000000 1   1      INT error;
  601.    000000 1   1
  602.    000000 1   1      STRING
  603.    000000 1   1          .srec := @rec '<<' 1;
  604.    000000 1   1
  605.    000000 1   1      ! increment key value past current value.
  606.    000000 1   1      srec [ db^rec^key^off + db^rec^key^len - 1 ] :=
  607.    000004 1   1         srec [ db^rec^key^off + db^rec^key^len - 1 ] '+' 1;
  608.    000010 1   1
  609.    000010 1   1      CALL KEYPOSITION ( db^fnum, rec );
  610.    000020 1   1      CALL READ ( db^fnum, result, db^rec^len, result^len );
  611.    000031 1   1      CALL FILEINFO ( db^fnum, error );
  612     000040 1   1      RETURN error;
  613.    000042 1   1    END; ! next^record^request.

ERROR                          Variable        INT        L+001    Direct
REC                            Variable        INT        L-005    Indirect
RESULT                         Variable        INT        L-004    Indirect
RESULT^LEN                     Variable        INT        L-003    Indirect
SREC                           Variable        STRING     L+002    Indirect

00000  002001 170705 030001 024700 103027 153402 003001 157402     00010 040011 170705 030001 024711 002003 100030 024700 027
00020  040011 170704 005001 170703 000002 024755 100036 024700     00030 027000 040011 070401 024711 002013 005030 024700 027
00040  040401 125006
```

```
615.  000000 0  0  ! this procedure is used to process a request.
616.  000000 0  0
617.  000000 0  0  PROC process^user^request ( recv^buf, recv^cnt, pid );
618.  000000 1  0     INT .recv^buf,
619.  000000 1  0         recv^cnt,
620.  000000 1  0        .pid; ! process id of requestor.
621.  000000 1  0
622.  000000 1  0    BEGIN
623.  000000 1  1      INT  status,
624.  000000 1  1           requestor,      ! directory entry no. of current requestor.
625.  000000 1  1           reply^len := 0; ! reply length for current request.
626.  000000 1  1
627.  000000 1  1      ! get requestor number of current requestor.
628.  000000 1  1      IF NOT ( requestor := lookuppid ( pid ) ) THEN ! invalid requestor.
629.  000010 1  1        BEGIN
630.  000010 1  2          CALL REPLY ( ,,,, 60 ); ! return "device has been downed" error.
631.  000015 1  2          RETURN;
632.  000016 1  2        END;
633.  000016 1  1
634.  000016 1  1      ! check for duplicate request.
635.  000016 1  1      IF recv^buf [ sync ] <> sync^count[ requestor ] THEN ! not duplicate, nonretryable.
636.  000023 1  1        BEGIN
637.  000023 1  2
638.  000023 1  2          IF recv^buf [ request^type ] <= 1 THEN ! nonretryable, so checkpoint current state to backup.
639.  000027 1  2            BEGIN
640.  000027 1  3              ! save sync count of current requestor.
641.  000027 1  3              sync^count [ requestor ] := recv^buf;
642.  000031 1  3
643.  000031 1  3              IF ( status := CHECKPOINT ( stack^base,                    ! *** CHECKPOINT *** !
644.  000031 1  3                                          ,db^fnum,
645.  000031 1  3                                          sync^count [ requestor ], 1,
646.  000031 1  3                                          reply^error [ old^requestor ], 1,
647.  000031 1  3                                          recv^buf, ( recv^cnt + 1 ) / 2 ) ) THEN
648.  000060 1  3                 CALL analyze^checkpoint^status ( status );
649.  000063 1  3
650.  000063 1  3              ! save requestor number of current requestor for a subsequent checkpoint.
651.  000063 1  3              old^requestor := requestor;
652.  000065 1  3            END; ! of checkpoint.
653.  000065 1  2
654.  000065 1  2          ! save request^type.
655.  000065 1  2          reply^buf ':=' recv^buf FOR 2;
656.  000071 1  2
657.  000071 1  2          ! process the data base request.
658.  000071 1  2          CASE recv^buf [ request^type ] OF
659.  000074 1  2            BEGIN
660.  000074 1  3
661.  000074 1  3              ! "insert" request.
662.  000074 1  3              reply^error [ requestor ] := insert^request ( recv^buf [ record ] );
663.  000103 1  3
664.  000103 1  3              ! "delete" request.
665.  000103 1  3              reply^error [ requestor ] := delete^request ( recv^buf [ record ] );
666.  000112 1  3
667.  000112 1  3              ! "query" request.
668.  000112 1  3              reply^error [ requestor ] :=
669.  000112 1  3                 query^request ( recv^buf [ record ], reply^buf [ record ], reply^len );
670.  000123 1  3
671.  000123 1  3              ! "next entry" request.
```

```
  672.    000123 1  3              reply^error [ requestor ] :=
  673.    000123 1  3                 next^record^request ( recv^buf [ record ], reply^buf [ record ], reply^len );
  674.    000134 1  3
  675.    000134 1  3              OTHERWISE reply^error [ requestor ] := 29; ! bad param.
  676.    000137 1  3            END;
  677.    000154 1  2          END;
  678.    000154 1  1
  679.    000154 1  1        ! return the reply to the requestor.
  680.    000154 1  1        CALL REPLY ( reply^buf, reply^len + 4, , , reply^error [ requestor ] );
  681.    000165 1  1
  682.    000165 1  1    END; ! process^user^request.

PID                             Variable        INT        L-003    Indirect
RECV^BUF                        Variable        INT        L-005    Indirect
RECV^CNT                        Variable        INT        L-004    Direct
REPLY^LEN                       Variable        INT        L+003    Direct
REQUESTOR                       Variable        INT        L+002    Direct
STATUS                          Variable        INT        L+001    Direct

00000  002002 100000 024700 170703 024700 027000 034402 015406     00010 002004 100074 100001 024711 027000 125006 140705 033
00020  143032 000215 012131 102001 142705 001001 011036 140705     00030 147032 170007 100000 040011 173032 100001 024744 031
00040  171033 100001 170705 040704 104001 100002 000213 024733     00050 002022 005005 004374 100000 024711 027000 034401 014
00060  040401 024700 027000 040402 044031 170034 170705 100002     00070 026007 103001 143705 010444 103002 173705 024700 027
00100  033402 147033 010451 103002 173705 024700 027000 033402     00110 147033 010442 103002 173705 173034 070403 024722 027
00120  033402 147033 010431 103002 173705 173034 070403 024722     00130 027000 033402 147033 010420 033402 100035 147033 010
00140  100003 000205 011002 000100 010401 100004 000030 177725     00150 177733 177741 177751 177761 170034 040403 104004 000
00160  033402 143033 100031 024755 027000 125006
```

```
  684.    000000 0  0  ! this is the main execution loop of the server process.  The server waits
  685.    000000 0  0  ! for incoming requests or system messages.
  686.    000000 0  0
  687.    000000 0  0  PROC execute;
  688.    000000 1  0    BEGIN
  689.    000000 1  1
  690.    000000 1  1      INT .pid[0:3],       ! requestor <process id>.
  691.    000000 1  1          system^message,
  692.    000000 1  1          status;
  693.    000000 1  1
  694.    000000 1  1      WHILE 1 DO ! loop on requests.
  695.    000003 1  1        BEGIN
  696.    000003 1  2
  697.    000003 1  2          ! read $RECEIVE file.
  698.    000003 1  2          CALL READUPDATE ( recv^fnum, recv^buf, message^len );
  699.    000014 1  2          CALL AWAITIO ( recv^fnum,, recv^cnt );
  700.    000024 1  2          IF >= THEN ! read a message.
  701.    000025 1  2            BEGIN
  702.    000025 1  3              system^message := >; ! save system message condition.
  703.    000032 1  3              CALL LASTRECEIVE ( pid );
  704.    000037 1  3              IF system^message THEN
  705.    000041 1  3                CALL analyze^system^message ( recv^buf, recv^cnt, pid )
  706.    000046 1  3              ELSE
  707.    000047 1  3                CALL process^user^request ( recv^buf, recv^cnt, pid );
  708.    000054 1  3            END;     ! read a message.
  709.    000054 1  2        END;     ! loop on requests.
  710.    000055 1  1    END; ! execute.

PID                             Variable        INT        L+001    Indirect
STATUS                          Variable        INT        L+003    Direct
SYSTEM^MESSAGE                  Variable        INT        L+002    Direct


00000  070404 024700 002006 040010 170026 005001 004004 024722     00010 002003 100034 024700 027000 070010 100000 070027 024
00020  002003 100024 024700 027000 014027 016002 100777 010401     00030 100000 044402 170401 100000 100002 024722 027000 040
00040  014406 170026 040027 170401 024722 027000 010405 170026     00050 040027 170401 024722 027000 010726 125003
```

```
  712.    000000 0  0  ! this is the "main".  This is where the primary/backup determination is
  713.    000000 0  0  ! made.
  714.    000000 0  0
  715.    000000 0  0  PROC serve MAIN;
  716.    000000 1  0
  717.    000000 1  0    BEGIN
  718.    000000 1  1
  719.    000000 1  1      INT  base = 'L' + 1,
  720.    000000 1  1           .ppdentry[0:8];
  721.    000000 1  1
  722.    000000 1  1      ! save stack^base for checkpointing.
  723.    000000 1  1      @stack^base := @base;
  724.    000005 1  1
  725.    000005 1  1      CALL ARMTRAP ( 0, -1 );
  726.    000011 1  1
  727.    000011 1  1      ! get process name.
  728.    000011 1  1      CALL GETCRTPID ( MYPID, ppdentry );
  729.    000015 1  1      CALL LOOKUPPROCESSNAME ( ppdentry );
  730.    000020 1  1      IF < THEN CALL ABEND;   ! not named.
  731.    000024 1  1
  732.    000024 1  1      ! calculate backup cpu number.  cpu's are paired 0-1, 2-3, 4-5, ...
  733.    000024 1  1      backup^cpu := MYPID.<0:7>;
  734.    000027 1  1      backup^cpu.<15> := NOT backup^cpu.<15>;
  735.    000040 1  1
  736.    000040 1  1      ! monitor all cpus.
  737.    000040 1  1      CALL MONITORCPUS ( -1 );
  738.    000043 1  1
  739.    000043 1  1      IF NOT ppdentry[4] THEN ! im the primary.
  740.    000046 1  1        BEGIN
  741.    000046 1  2          CALL read^start^up^message;
  742.    000047 1  2          CALL open^primarys^files;
  743.    000050 1  2          CALL create^backup ( backup^cpu );
  744.    000053 1  2          CALL execute;
  745.    000054 1  2        END
  746.    000054 1  1      ELSE                    ! im the backup.
  747.    000055 1  1        BEGIN
  748.    000055 1  2          ! wait for failure
  749.    000055 1  2          CALL CHECKMONITOR;
  750.    000057 1  2          CALL ABEND;
  751.    000062 1  2        END;
  752.    000062 1  1    END; ! serve.

BASE                          Variable        INT        L+001    Direct
PPDENTRY                      Variable        INT        L+001    Indirect


00000  070402 024700 002011 070401 044007 100000 100777 024711     00010 027000 027000 170401 024711 027000 170401 024700 027
00020  013003 000002 024711 027000 027000 030110 044000 040000     00030 006001 015402 100777 010401 100000 100001 070000 000
00040  100777 024700 027000 103004 143401 015407 027000 027000     00050 040000 024700 027000 027000 010405 027000 000107 000
00060  024711 027000 000002 024711 127000
```

ABEND	Proc			
ADDPID	Proc	INT		
ANALYZE^CHECKPOINT^STATUS	Proc			
ANALYZE^SYSTEM^MESSAGE	Proc			
ARMTRAP	Proc			
AWAITIO	Proc			
BACKUP^CPU	Variable	INT	G+000	Direct
BACKUP^PID	Variable	INT	G+001	Direct
BACKUP^UP	Variable	INT	G+006	Direct
BUF	Variable	INT	G+030	Indirect
CHECKMONITOR	Proc	INT		
CHECKOPEN	Proc			
CHECKPOINT	Proc	INT		
CREATE^BACKUP	Proc			
DB^FLAGS	Literal			%000040
DB^FNAME	Variable	INT	G+012	Direct
DB^FNUM	Variable	INT	G+011	Direct
DB^REC^KEY^LEN	Literal			%000030
DB^REC^KEY^OFF	Literal			%000000
DB^REC^LEN	Literal			%000400
DB^SYNC^DEPTH	Literal			%000001
DEBUG	Proc			
DELALLPIDS	Proc			
DELETE^REQUEST	Proc	INT		
DELPID	Proc			
EXECUTE	Proc			
FILEINFO	Proc			
GETCRTPID	Proc			
INSERT^REQUEST	Proc	INT		
KEYPOSITION	Proc			
LASTRECEIVE	Proc			
LOOKUPPID	Proc	INT		
LOOKUPPROCESSNAME	Proc			
MAX^REQSTRS	Literal			%000020
MESSAGE^LEN	Literal			%000404
MESSAGE^SIZE	Literal			%000202
MOM	Proc			
MONITORCPUS	Proc			
MYPID	Proc	INT		
NEWPROCESS	Proc			
NEXT^RECORD^REQUEST	Proc	INT		
NO^WAIT	Literal			%000001
OLD^REQUESTOR	Variable	INT	G+031	Direct
OPEN	Proc			
OPEN^BACKUPS^FILES	Proc			
OPEN^MSGS	Literal			%040000
OPEN^PRIMARYS^FILES	Proc			
PIDS	Variable	INT	G+035	Indirect
PROCESS^USER^REQUEST	Proc			
PROGRAMFILENAME	Proc			
PROTECTED	Literal			%000040
QUERY^REQUEST	Proc	INT		
READ	Proc			
READUPDATE	Proc			
READ^START^UP^MESSAGE	Proc			
RECORD	Literal			%000002

RECV^BUF	Variable	INT	G+026	Indirect
RECV^CNT	Variable	INT	G+027	Direct
RECV^FLAGS	Literal			%040001
RECV^FNUM	Variable	INT	G+010	Direct
RECV^SYNC^DEPTH	Literal			%000001
REPLY	Proc			
REPLY^BUF	Variable	INT	G+034	Indirect
REPLY^ERROR	Variable	INT	G+033	Indirect
REQUEST^TYPE	Literal			%000001
SERVE	Proc			
STACK^BASE	Variable	INT	G+007	Indirect
STOP	Proc			
STOP^COUNT	Variable	INT	G+005	Direct
SYNC	Literal			%000000
SYNC^COUNT	Variable	INT	G+032	Indirect
WRITE	Proc			
WRITEUPDATE	Proc			

PEP	BASE	LIMIT	ENTRY	ATTRIBUTES	NAME
002	000512	000600	000516		ADDPID
003	000370	000453	000370		ANALYZE^CHECKPOINT^STATUS
004	000733	001147	000733		ANALYZE^SYSTEM^MESSAGE
005	000311	000367	000311		CREATE^BACKUP
006	000662	000732	000662		DELALLPIDS
007	001227	001260	001227		DELETE^REQUEST
010	000601	000661	000601		DELPID
011	001511	001566	001511		EXECUTE
012	001205	001226	001205		INSERT^REQUEST
013	000454	000511	000454		LOOKUPPID
014	001261	001322	001261		NEXT^RECORD^REQUEST
015	000223	000310	000223		OPEN^BACKUPS^FILES
016	000205	000222	000205		OPEN^PRIMARYS^FILES
017	001323	001510	001323		PROCESS^USER^REQUEST
020	001150	001204	001150		QUERY^REQUEST
021	000023	000204	000023		READ^START^UP^MESSAGE
022	001567	001653	001567	M	SERVE

PAGE 26 $BOOKS1.M096B01.SERVESRC [1] LOAD MAP 2/23/82 13:

PEP	BASE	LIMIT	ENTRY	ATTRIBUTES	NAME
021	000023	000204	000023		READ^START^UP^MESSAGE
016	000205	000222	000205		OPEN^PRIMARYS^FILES
015	000223	000310	000223		OPEN^BACKUPS^FILES
005	000311	000367	000311		CREATE^BACKUP
003	000370	000453	000370		ANALYZE^CHECKPOINT^STATUS
013	000454	000511	000454		LOOKUPPID
002	000512	000600	000516		ADDPID
010	000601	000661	000601		DELPID
006	000662	000732	000662		DELALLPIDS
004	000733	001147	000733		ANALYZE^SYSTEM^MESSAGE
020	001150	001204	001150		QUERY^REQUEST
012	001205	001226	001205		INSERT^REQUEST
007	001227	001260	001227		DELETE^REQUEST
014	001261	001322	001261		NEXT^RECORD^REQUEST
017	001323	001510	001323		PROCESS^USER^REQUEST
011	001511	001566	001511		EXECUTE
022	001567	001653	001567	M	SERVE

```
Object file name is $BOOKS1.M096B01.serveobj
This object file will run on either a TNS or a TNS/II
Number of errors = 0
Number of warnings = 0
Primary global storage=30
Secondary global storage=503
Code size=921
Data area size=2  pages
Code area size=1  pages
Maximum symbol table space available = 24892, used = 1298
Maximum extended symbol table space available =     0, used = 0
Number of source lines=2142
Elapsed time -  00:00:39
```

APPENDIX A

PROCEDURE SYNTAX SUMMARY

The list on the following pages gives the calling syntax of the GUARDIAN operating system procedures and related Tandem software procedures callable by user programs. For each procedure, a reference is given to the manual or manuals that explain how to use it.

Procedures available only on NonStop systems are marked "(I only)". Procedures available only on NonStop II systems are marked "(II only)".

CALL ABEND

See "Process Control" section of the GUARDIAN Operating System Programming Manual.

CALL ACTIVATEPROCESS (<process id>)

See "Process Control" section of the GUARDIAN Operating System Programming Manual.

```
<status> := ALLOCATESEGMENT ( <segment id>              (II only)
                            , <segment size>
                            , <file name>
                            , <pin> )
```

See "Memory Management Procedures" section of the GUARDIAN Operating System Programming Manual.

→

CALL ALTERPRIORITY (<process id> , <priority>)

See "Process Control" section of the GUARDIAN Operating System Programming Manual.

CALL ARMTRAP (<trap label> , <trap address>)

See "Traps" section of the GUARDIAN Operating System Programming Manual.

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

See "File System Procedures" section of the GUARDIAN Operating System Programming Manual, the ENSCRIBE Programming Manual, the ENVOY Byte-Oriented Protocols Reference Manual, or the ENVOYACP Bit-Oriented Protocols Reference Manual.

```
! INT:function ! BLINK^SCREEN ( @<screen name>
                              , <buffer>
                              , <field name>
                              , <blink> )
```

See "Entry Procedures" section of ENTRY Screen Formatter Operating and Programming Manual.

CALL CANCEL (<file number>)

See "File System Procedures" section of the GUARDIAN Operating System Programming Manual, the ENSCRIBE Programming Manual, the ENVOY Byte-Oriented Protocols Reference Manual, or the ENVOYACP Bit-Oriented Protocols Reference Manual.

CALL CANCELREQ (<file number> , <tag>)

See "File System Procedures" section of the GUARDIAN Operating System Programming Manual, the ENSCRIBE Programming Manual, or the ENVOYACP Bit-Oriented Protocols Reference Manual.

CALL CANCELTIMEOUT (<tleaddress>) (II only)

See "Process Control" section of the GUARDIAN Operating System Programming Manual.

→

```
CALL CHANGELIST ( <file number> , <function> , <parameter> )
```

See "File System Procedures" section of the ENVOY Byte-Oriented Protocols Reference Manual or the ENVOYACP Bit-Oriented Protocols Reference Manual.

```
<state> := CHECK^BREAK ( { <common fcb> } )
                         { <file fcb>   }
```

See "Sequential I/O Procedures" section of the GUARDIAN Operating System Programming Manual.

```
CALL CHECKCLOSE ( <file number>
                  , <tape disposition> )
```

See "Checkpointing Facility" section of the GUARDIAN Operating System Programming Manual.

```
<retval> := CHECK^FILE ( { <common fcb> } , <operation> )
                         { <file fcb>   }
```

See "Sequential I/O Procedures" section of the GUARDIAN Operating System Programming Manual.

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

See "Checkpointing Facility" section of the GUARDIAN Operating System Programming Manual.

```
CALL CHECKOPEN ( <file name>
                 , <file number>
                 , <flags>
                 , <sync or receive depth>
                 , <sequential block buffer>
                 , <buffer length>
                 , <back error> )
```

See "Checkpointing Facility" section of the GUARDIAN Operating System Programming Manual.

→

```
{ <status> := } CHECKPOINT ( <stack base>
{ CALL       }               , <buffer 1>
                             , <count 1>
                             , <buffer 2>
                             , <count 2>
                                  .
                                  .
                                  .
                             , <buffer 13>
                             , <count 13> )
```

See "Checkpointing Facility" section of the GUARDIAN Operating System Programming Manual.

```
{ <status> := } CHECKPOINTMANY ( <stack base>
{ CALL       }                   , <descriptors> )
```

See "Checkpointing Facility" section of the GUARDIAN Operating System Programming Manual.

```
! INT:function ! CHECK^SCREEN ( @<screen name>
                               , SCREEN
                               , <buffer>
                               , <check procedure> )
```

See "Entry Procedures" section of ENTRY Screen Formatter Operating and Programming Manual.

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

See "Checkpointing Facility" section of the GUARDIAN Operating System Programming Manual.

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

See "File System Procedures" section of the GUARDIAN Operating System Programming Manual, the ENSCRIBE Programming Manual, the ENVOY Byte-Oriented Protocols Reference Manual, or the ENVOYACP Bit-Oriented Protocols Reference Manual.

```
{ CALL       } CLOSE^FILE ( { <common fcb> }
{ <error> := }              { <file fcb>   }

                           , <tape disposition> )
```

See "Sequential I/O Procedures" section of the GUARDIAN Operating System Programming Manual.

→

```
CALL CONTIME ( <date and time>
             , <t0>
             , <t1>
             , <t2> )
```

See "Utility Procedures" section of the GUARDIAN Operating System Programming Manual.

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

See "File System Procedures" section of the GUARDIAN Operating System Programming Manual, the ENSCRIBE Programming Manual, the ENVOY Byte-Oriented Protocols Reference Manual, or the ENVOYACP Bit-Oriented Protocols Reference Manual.

```
CALL CONTROLBUF ( <file number> , <operation>
                , <buffer> , <count>
                , <count transferred>
                , <tag> )
```

See "File System Procedures" section of the GUARDIAN Operating System Programming Manual.

```
CALL CONVERTPROCESSNAME ( <process name> )
```

See "Process Control" section of the GUARDIAN Operating System Programming Manual, or see the EXPAND Users Manual.

```
CALL CREATE ( <file name>
            , <primary extent size>
            , <file code>
            , <secondary extent size>
            , <file type>
            , <record length>
            , <data block length>
            , <key-sequenced params>
            , <alternate key params>
            , <partition params>       )
```

See "File System Procedures" section of the GUARDIAN Operating System Programming Manual or the ENSCRIBE Programming Manual.

→

CALL CREATEPROCESSNAME (<process name>)

See "Process Control" section of the GUARDIAN Operating System Programming Manual.

CALL CREATEREMOTENAME (<name> , <system number>)

See "Process Control" section of the GUARDIAN Operating System Programming Manual, or see the EXPAND Users Manual.

<accessor id> := CREATORACCESSID

See "Security System" section of the GUARDIAN Operating System Programming Manual.

CALL DEALLOCATESEGMENT (<segment id> , <flags>) (II only)

See "Memory Management Procedures" section of the GUARDIAN Operating System Programming Manual.

CALL DEBUG

See "Utility Procedures" section of the GUARDIAN Operating System Programming Manual.

CALL DEFINELIST (<file number> , <address list>
, <address size> , <num entries>
, <polling count> , <polling type>)

See "File System Procedures" section of the ENVOY Byte-Oriented Protocols Reference Manual or the ENVOYACP Bit-Oriented Protocols Reference Manual.

<status> := DEFINEPOOL (<pool head> , <pool> , <pool size>) (II only)

See "Memory Management Procedures" section of the GUARDIAN Operating System Programming Manual.

CALL DELAY (<time period>)

See "Process Control" section of the GUARDIAN Operating System Programming Manual.

→

```
CALL DEVICEINFO ( <file name>
                , <device type>
                , <physical record length> )
```

See "File System Procedures" section of the GUARDIAN Operating System Programming Manual, the ENSCRIBE Programming Manual, the ENVOY Byte-Oriented Protocols Reference Manual, the ENVOYACP Bit-Oriented Protocols Reference Manual, or the AXCESS Data Communications Programming Manual.

```
<status> := EDITREAD ( <edit control block> , <buffer>
                     , <buffer length> , <sequence number> )
```

See "File System Procedures" section of the GUARDIAN Operating System Programming Manual or the ENSCRIBE Programming Manual.

```
<status> := EDITREADINIT ( <edit control block> , <file number>
                         , <buffer length> )
```

See "File System Procedures" section of the GUARDIAN Operating System Programming Manual or the ENSCRIBE Programming Manual.

```
CALL ENFORMFINISH ( <ctlblock> )
```

See the ENFORM Reference Manual.

```
{ <count> := } ENFORMRECEIVE ( <ctlblock> , <buffer> )
{ CALL       }
```

See the ENFORM Reference Manual.

```
CALL ENFORMSTART ( <ctlblock>
                 , <compiled physical filename>
                 , <buffer length>
                 , <error number>
                 , <restart flag>
                 , <param list>
                 , <assign list>
                 , <process name>
                 , <cpu>
                 , <priority>
                 , <timeout> )
```

See the ENFORM Reference Manual.

→

```
! INT:function ! EXPAND^SCREEN ( @<screen name> , SCREEN
                                , <buffer>
                                , <rewrite form> )
```

See "Entry Procedures" section of ENTRY Screen Formatter Operating and Programming Manual.

```
<status> := FILEERROR ( <file number> )
```

See "File System Procedures" section of the GUARDIAN Operating System Programming Manual or the ENSCRIBE Programming Manual.

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

See "File System Procedures" section of the GUARDIAN Operating System Programming Manual, the ENSCRIBE Programming Manual, the ENVOY Byte-Oriented Protocols Reference Manual, or the ENVOYACP Bit-Oriented Protocols Reference Manual.

```
CALL FILERECINFO ( <file number>
                 , <current key specifier>
                 , <current key value>
                 , <current key length>
                 , <current primary key value>
                 , <current primary key length>
                 , <partition in error>
                 , <specifier of key in error>
                 , <file type>
                 , <logical record length>
                 , <block length>
                 , <key-sequenced params>
                 , <alternate key params>
                 , <partition params> )
```

See "File System Procedures" section of the ENSCRIBE Programming Manual.

→

```
CALL FIXSTRING ( <template> , <template length>
                 , <data> , <data length>
                 , <maximum data length>
                 , <modification status> )
```

See "Utility Procedures" section of the GUARDIAN Operating System Programming Manual.

```
! INT:function ! FL^SCREEN ( <field name> )
```

See ENTRY Screen Formatter Operating and Programming Manual.

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

See "File System Procedures" section of the GUARDIAN Operating System Programming Manual or the ENSCRIBE Programming Manual, or see the EXPAND Users Manual.

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

See "File System Procedures" section of the GUARDIAN Operating System Programming Manual or the ENSCRIBE Programming Manual, or see the EXPAND Users Manual.

```
{ <length> := } FNAMEEXPAND ( <external file name>
{ CALL       }                , <internal file name>
                              , <default names> )
```

See "File System Procedures" section of the GUARDIAN Operating System Programming Manual or the ENSCRIBE Programming Manual, or see the EXPAND Users Manual.

```
{ <status> := } FORMATCONVERT ( <iformat>
{  CALL      }                  , <iformatlen>
                                , <eformat>
                                , <eformatlen>
                                , <scales>
                                , <scalecount>
                                , <conversion> )
```

See "Formatter" section of the GUARDIAN Operating System Programming Manual.

→

```
{ <error> := } FORMATDATA ( <buffer>
{  CALL      }              , <bufferlen>
                            , <bufferoccurs>
                            , <length>
                            , <iformat>
                            , <variablelist>
                            , <variablelistlength>
                            , <flags> )
```

See "Formatter" section of the GUARDIAN Operating System Programming Manual.

```
CALL GETCRTPID ( <cpu, pin>
               , <process id> )
```

See "Process Control" section of the GUARDIAN Operating System Programming Manual.

```
<status> := GETDEVNAME ( <logical device no>
                       , <device name>
                       , <system number> )
```

See "File System Procedures" section of the GUARDIAN Operating System Programming Manual or the ENSCRIBE Programming Manual.

```
<address> := GETPOOL ( <pool head> , <block size> )     (II only)
```

See "Memory Management Procedures" section of the GUARDIAN Operating System Programming Manual.

```
CALL GETPPDENTRY ( <entry number> , <system number>
                 , <PPD entry>  )
```

See "Process Control" section of the GUARDIAN Operating System Programming Manual, or see the EXPAND Users Manual.

```
CALL GETREMOTECRTPID ( <pid> , <process id> , <system number> )
```

See "Process Control" section of the GUARDIAN Operating System Programming Manual, or see the EXPAND Users Manual.

→

```
CALL GETSYNCINFO ( <file number>
                 , <sync block>
                 , <sync block size> )
```

See "Checkpointing Facility" section of the GUARDIAN Operating System Programming Manual.

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

See "File System Procedures" section of the GUARDIAN Operating System Programming Manual, or see the EXPAND Users Manual. Manual.

```
CALL GIVE^BREAK ( { <common fcb> } )
                  { <file fcb>   }
```

See "Sequential I/O Procedures" section of the GUARDIAN Operating System Programming Manual.

```
CALL HALTPOLL ( <file number> )
```

See "File System Procedures" section of the ENVOY Byte- Oriented Protocols Reference Manual or the ENVOYACP Bit-Oriented Protocols Reference Manual.

```
CALL HEAPSORT ( <array> , <num elements> , <size of element>
              , <compare proc> )
```

See "Utility Procedures" section of the GUARDIAN Operating System Programming Manual.

```
{ <status> := } INITIALIZER ( <rucb>
{  CALL     }                , <passthru>
                             , <startupproc>
                             , <paramsproc>
                             , <assignproc>
                             , <flags>       )
```

See "Utility Procedures" section of the GUARDIAN Operating System Programming Manual.

→

```
CALL KEYPOSITION ( <file number>
                 , <key value>
                 , <key specifier>
                 , <length word>
                 , <positioning mode>  )
```

See "File System Procedures" section of the ENSCRIBE Programming Manual.

```
<last address> := LASTADDR
```

See "Utility Procedures" section of the GUARDIAN Operating System Programming Manual.

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

See "File System Procedures" section of the GUARDIAN Operating System Programming Manual.

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

See "File System Procedures" section of the GUARDIAN Operating System Programming Manual, or see the EXPAND Users Manual.

```
{ <state> := } LOCKDATA ( <address>                    (I only)
{ CALL      }           , <count>
                        , <sys map> )
```

See "Advanced Memory Management" section of the GUARDIAN Operating System Programming Manual.

```
CALL LOCKFILE ( <file number>
              , <tag> )
```

See "File System Procedures" section of the GUARDIAN Operating System Programming Manual or the ENSCRIBE Programming Manual.

→

```
<status> := LOCKMEMORY ( <address>                          (II only)
                       , <byte count>
                       , <timeoutvalue>
                       , <parameter1>
                       , <parameter2> )
```

See "Advanced Memory Management" section of the GUARDIAN Operating System Programming Manual.

```
CALL LOCKREC ( <file number>
             , <tag> )
```

See "File System Procedures" section of the ENSCRIBE Programming Manual.

```
CALL LOOKUPPROCESSNAME ( <ppd entry> )
```

See "Process Control" section of the GUARDIAN Operating System Programming Manual.

```
CALL MOM ( <process id> )
```

See "Process Control" section of the GUARDIAN Operating System Programming Manual.

```
CALL MONITORCPUS ( <cpu mask> )
```

See "Checkpointing Facility" section of the GUARDIAN Operating System Programming Manual.

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

See "File System Procedures" section of the GUARDIAN Operating System Programming Manual, or see the EXPAND Users Manual.

```
CALL MONITORNEW ( <enable> )                                (II only)
```

See "File System Procedures" section of the GUARDIAN Operating System Programming Manual.

→

<my cpu,pin> := MYPID

See "Process Control" section of the GUARDIAN Operating System Programming Manual.

<system number> := MYSYSTEMNUMBER

See "Process Control" section of the GUARDIAN Operating System Programming Manual, or see the EXPAND Users Manual.

CALL MYTERM (<file name>)

See "Process Control" section of the GUARDIAN Operating System Programming Manual, or see the EXPAND Users Manual.

```
CALL NEWPROCESS ( <filenames>
                , <priority>
                , <memory pages>
                , <processor>
                , <process id>
                , <error>
                , <name>          )
```

See "Process Control" section of the GUARDIAN Operating System Programming Manual.

```
CALL NEWPROCESSNOWAIT ( <filenames>                    (II only)
                      , <priority>
                      , <memory pages>
                      , <processor>
                      , <process id>
                      , <error>
                      , <name>         )
```

See "Process Control" section of the GUARDIAN Operating System Programming Manual.

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

See "File System Procedures" section of the GUARDIAN Operating System Programming Manual or the ENSCRIBE Programming Manual.

→

```
{ CALL          } NO^ERROR ( <state> , <file fcb>
{ <no retry> := }          , <good error list> , <retryable> )
```

See "Sequential I/O Procedures" section of the GUARDIAN Operating System Programming Manual.

```
{ <next address> := } NUMIN ( <ascii number>
{ CALL              }       , <signed result>
                            , <base>
                            , <status> )
```

See "Utility Procedures" section of the GUARDIAN Operating System Programming Manual.

```
CALL NUMOUT ( <ascii result>
            , <unsigned integer>
            , <base>
            , <width> )
```

See "Utility Procedures" section of the GUARDIAN Operating System Programming Manual.

```
CALL OPEN ( <file name>
          , <file number>
          , <flags>
          , <sync or receive depth>
          , <primary file number> , <primary process id>
          , <sequential block buffer> , <buffer length> )
```

See "File System Procedures" section of the GUARDIAN Operating System Programming Manual, the ENSCRIBE Programming Manual, the ENVOY Byte-Oriented Protocols Reference Manual, or the ENVOYACP Bit-Oriented Protocols Reference Manual.

```
{  CALL     } OPEN^FILE ( <common fcb> , <file fcb>
{ <error> := }           , <block buffer>
                         , <block buffer length>
                         , <flags>
                         , <flags mask>
                         , <max record length>
                         , <prompt char>
                         , <error file fcb> )
```

See "Sequential I/O Procedures" section of the GUARDIAN Operating System Programming Manual.

→

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

See "File System Procedures" section of the GUARDIAN Operating System Programming Manual or the ENSCRIBE Programming Manual.

```
! INT:function ! POSITION^SCREEN ( @<screen name>
                                 , SCREEN
                                 , <buffer>
                                 , <field name> )
```

See "Entry Procedures" section of ENTRY Screen Formatter Operating and Programming Manual.

```
<error code> := PRINTCOMPLETE ( <file number to Supervisor>
                              , <print control buffer> )
```

See "Print Processes" section of the Spooler System Management Guide.

```
<error code> := PRINTINFO ( <job buffer> , <copies remaining>
                          , <current page>, <current line>
                          , <lines printed> )
```

See "Print Processes" section of the Spooler System Management Guide.

```
<error code> := PRINTINIT ( <file number to Supervisor>
                          , <print control buffer> )
```

See "Print Processes" section of the Spooler System Management Guide.

```
<error code> := PRINTREAD ( <job buffer>
                          , <data line> , <read count>
                          , <count read> , <page number> )
```

See "Print Processes" section of the Spooler System Management Guide.

→

```
<error code> := PRINTREADCOMMAND ( <print control buffer>
                                 , <control number> , <device>
                                 , <dev flags> , <dev param>
                                 , <dev width> , <skip num>
                                 , <data file> , <job number>
                                 , <location> , <form name>
                                 , <report name> , <pagesize> )
```

See "Print Processes" section of the Spooler System Management Guide.

```
<error code> := PRINTSTART ( <job buffer>
                           , <print control buffer>
                           , <data file number>  )
```

See "Print Processes" section of the Spooler System Management Guide.

```
<error code> := PRINTSTATUS ( <file number to Supervisor>
                            , <print control buffer>
                            , <message type> , <device>
                            , <error> , <num copies>
                            , <page> , <line>
                            , <lines printed> )
```

See "Print Processes" section of the Spooler System Management Guide.

```
<last priority> := PRIORITY ( <priority> )
```

See "Process Control" section of the GUARDIAN Operating System Programming Manual.

```
<accessor id> := PROCESSACCESSID
```

See "Security System" section of the GUARDIAN Operating System Programming Manual.

→

```
{ <error> := } PROCESSINFO ( <cpu,pin>
{ CALL       }               , <process id>
                             , <creator accessor id>
                             , <process accessor id>
                             , <priority>
                             , <program file name>
                             , <home terminal>
                             , <system number>
                             , <search mode> )
```

See "Process Control" section of the GUARDIAN Operating System Programming Manual, or see the EXPAND Users Manual.

<processor status> := PROCESSORSTATUS

See "Checkpointing Facility" section of the GUARDIAN Operating System Programming Manual, or see the EXPAND Users Manual.

CALL PROGRAMFILENAME (<program file>)

See "Process Control" section of the GUARDIAN Operating System Programming Manual.

CALL PURGE (<file name>)

See "File System Procedures" section of the GUARDIAN Operating System Programming Manual or the ENSCRIBE Programming Manual.

CALL PUTPOOL (<pool head> , <address>) (II only)

See "Memory Management Procedures" section of the GUARDIAN Operating System Programming Manual.

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

See "File System Procedures" section of the GUARDIAN Operating System Programming Manual, the ENSCRIBE Programming Manual, the ENVOY Byte-Oriented Protocols Reference Manual, or the ENVOYACP Bit-Oriented Protocols Reference Manual.

→

```
{ CALL       } READ^FILE ( <file fcb> , <buffer> , <count read>
{ <error> := }             , <prompt count>
                           , <max read count>
                           , <no wait> )
```

See "Sequential I/O Procedures" section of the GUARDIAN Operating System Programming Manual.

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

See "File System Procedures" section of the ENSCRIBE Programming Manual.

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

See "File System Procedures" section of the GUARDIAN Operating System Programming Manual or the ENSCRIBE Programming Manual.

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

See "File System Procedures" section of the ENSCRIBE Programming Manual.

```
CALL RECEIVEINFO ( <process id>
                 , <message tag>
                 , <sync id>
                 , <file number>
                 , <read count> )
```

See "File System Procedures" section of the GUARDIAN Operating System Programming Manual.

→

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

See "File System Procedures" section of the GUARDIAN Operating System Programming Manual or the ENSCRIBE Programming Manual.

```
<status> := REMOTEPROCESSORSTATUS ( <system number> )
```

See "File System Procedures" section of the GUARDIAN Operating System Programming Manual, or see the EXPAND Users Manual.

```
CALL RENAME ( <file number>
            , <new name> )
```

See "File System Procedures" section of the GUARDIAN Operating System Programming Manual or the ENSCRIBE Programming Manual.

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

See "File System Procedures" section of the GUARDIAN Operating System Programming Manual.

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

See "File System Procedures" section of the GUARDIAN Operating System Programming Manual or the ENSCRIBE Programming Manual.

```
CALL RESERVELCBS ( <no. receive lcbs>
                 , <no. send lcbs>  )
```

See "File System Advanced Features" section of the GUARDIAN Operating System Programming Manual.

```
CALL RESETSYNC ( <file number> )
```

See "Checkpointing Facility" section of the GUARDIAN Operating System Programming Manual.

→

```
CALL SAVEPOSITION ( <file number>
                  , <positioning block>
                  , <positioning block size> )
```

See "File System Procedures" section of the GUARDIAN Operating System Programming Manual or the ENSCRIBE Programming Manual.

```
{  CALL       } SET^FILE ( { <common fcb> } , <operation>
{ <error> := }             { <file fcb>   } , <new value>
                                             <old value> )
```

See "Sequential I/O Procedures" section of the GUARDIAN Operating System Programming Manual.

```
CALL SETLOOPTIMER ( <new time limit> , <old time limit> )
```

See "Process Control" section of the GUARDIAN Operating System Programming Manual.

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

See the "File System Procedures" section of the GUARDIAN Operating System Programming Manual, the ENSCRIBE Programming Manual, the ENVOY Byte-Oriented Protocols Reference Manual, or the ENVOYACP Bit-Oriented Protocols Reference Manual.

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

See the "File System Procedures" section of the GUARDIAN Operating System Programming Manual, the ENSCRIBE Programming Manual, the ENVOY Byte-Oriented Protocols Reference Manual, or the ENVOYACP Bit-Oriented Protocols Reference Manual.

```
CALL SETMYTERM ( <terminal name> )
```

See "Process Control" section of the GUARDIAN Operating System Programming Manual.

→

```
CALL SETPARAM ( <file number>
              , <function>
              , <param array>
              , <count>
              , <last param array>
              , <last count> )
```

See "Fundamental Programming Concepts" section of the AXCESS Data Communications Programming Manual, volume 1.

```
{ <last stop mode> := }  SETSTOP ( <stop mode> )
{  CALL               }
```

See "Process Control" section of the GUARDIAN Operating System Programming Manual.

```
CALL SETSYNCINFO ( <file number> , <sync block> )
```

See "Checkpointing Facility" section of the GUARDIAN Operating System Programming Manual.

```
CALL SHIFTSTRING ( <string> , <count> , <casebit> )
```

See "Utility Procedures" section of the GUARDIAN Operating System Programming Manual.

```
CALL SIGNALTIMEOUT ( <timeoutvalue>                    (II only)
                   , <parameter1>
                   , <parameter2>
                   , <tleaddress> )
```

See "Process Control" section of the GUARDIAN Operating System Programming Manual.

```
{ <num chars> := } SORTERROR ( <ctlblock> , <buffer>  )
{ CALL        }
```

See "Programmatic Mode" section of the SORT/MERGE Users Guide.

```
{ <dword error> := } SORTERRORDETAIL ( <ctlblock> )
{ CALL            }
```

See "Programmatic Mode" section of the SORT/MERGE Users Guide.

→

```
{ <error> := }  SORTMERGEFINISH ( <ctlblock>
{ CALL       }                    , <abort>
                                  , <spare 1>
                                  , <spare 2> )
```

See "Programmatic Mode" section of the SORT/MERGE Users Guide.

```
{ <error> := } SORTMERGERECEIVE ( <ctlblock>
{ CALL       }                    , <buffer>
                                  , <length>
                                  , <spare 1>
                                  , <spare 2> )
```

See "Programmatic Mode" section of the SORT/MERGE Users Guide.

```
{ <error> := } SORTMERGESEND ( <ctlblock>
{ CALL       }                   , <buffer>
                                 , <length>
                                 , <stream id>
                                 , <spare 1>
                                 , <spare 2>  )
```

See "Programmatic Mode" section of the SORT/MERGE Users Guide.

```
{ <error> := } SORTMERGESTART ( <ctlblock>
{ CALL       }                  , <key block>
                                , <number merge files>
                                , <number sort files>
                                , <in file name>
                                , <in file exclusion mode>
                                , <in file count>
                                , <in file length>
                                , <format>
                                , <out file name>
                                , <out file exclusion mode>
                                , <out file type>
                                , <flags>
                                , <errnum>
                                , <errproc>
                                , <scratch file name>
                                , <scratch block>
                                , <process start>
                                , <max record length>
                                , <collate sequence table>
                                , <spare 1>
                                , <spare 2>
                                , <spare 3>
                                , <spare 4>
                                , <spare 5> )
```

See "Programmatic Mode" section of the SORT/MERGE Users Guide.

→

```
{ <error> := }  SORTMERGESTATISTICS ( <ctlblock>
{ CALL       }                        , <length>
                                      , <statistics>
                                      , <spare 1>
                                      , <spare 2> )
```

See "Programmatic Mode" section of the SORT/MERGE Users Guide.

```
<error code> := SPOOLCONTROL ( <level 3 buffer>
                             , <operation>
                             , <parameter>
                             , <bytes written to buffer> )
```

See "Spooler Interface Procedures" section of the Spooler/ PERUSE Users Guide.

```
<error code> := SPOOLCONTROLBUF ( <level 3 buffer>
                                , <operation>
                                , <buffer>
                                , <count>
                                , <bytes written to buffer> )
```

See "Spooler Interface Procedures" section of the Spooler/ PERUSE Users Guide.

→

```
<error code> := SPOOLEND ( <level 3 buffer> , <flags> )
```

See "Spooler Interface Procedures" section of the Spooler/ PERUSE Users Guide.

```
<error code> := SPOOLERCOMMAND ( <file num to Supervisor>
                               , <keyword code>
                               , <keyword parameter>
                               , <subcommand code>
                               , <subcommand parameter>  )
```

See "Spooler Utility Procedures" section of the Spooler System Management Guide.

```
<error code> := SPOOLEREQUEST ( <supervisor file num>
                              , <job number> , <message> )
```

See "Spooler Utility Procedures" section of the Spooler System Management Guide.

```
<error code> := SPOOLERSTATUS ( <supervisor file num>
                              , <keyword code> , <scan type>
                              , <status buffer>  )
```

See "Spooler Utility Procedures" section of the Spooler System Management Guide.

```
<error code> := SPOOLJOBNUM ( <file number to collector>
                            , <job number> )
```

See "Spooler Interface Procedures" section of the Spooler/Peruse Users Guide.

```
<error code> := SPOOLSETMODE ( <level 3 buffer>
                             , <function>
                             , <parameter 1>
                             , <parameter 2>
                             , <bytes written to buffer>  )
```

See "Spooler Interface Procedures" section of the Spooler/ PERUSE Users Guide.

→

```
<error code> := SPOOLSTART ( <file number to collector>
                           , <level 3 buffer>
                           , <location>
                           , <form name>
                           , <report name>
                           , <number of copies>
                           , <page size>
                           , <flags> )
```

See "Spooler Interface Procedures" section of the Spooler/ PERUSE Users Guide.

```
<error code> := SPOOLWRITE ( <level 3 buffer>
                           , <print line>
                           , <write count>
                           , <bytes written to buffer> )
```

See "Spooler Interface Procedures" section of the Spooler/ PERUSE Users Guide.

```
CALL STEPMOM ( <process id> )
```

See "Process Control" section of the GUARDIAN Operating System Programming Manual.

```
CALL STOP ( <process id> )
```

See "Process Control" section of the GUARDIAN Operating System Programming Manual.

```
CALL SUSPENDPROCESS ( <process id> )
```

See "Process Control" section of the GUARDIAN Operating System Programming Manual.

```
CALL TAKE^BREAK ( <file fcb> )
```

See "Sequential I/O Procedures" section of the GUARDIAN Operating System Programming Manual.

```
CALL TIME ( <date and time> )
```

See "Utility Procedures" section of the GUARDIAN Operating System Programming Manual.

→

CALL TIMESTAMP (<interval clock>)

See "Utility Procedures" section of the GUARDIAN Operating System Programming Manual.

<version> := TOSVERSION

See "Utility Procedures" section of the GUARDIAN Operating System Programming Manual.

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

See "File System Procedures" section of the GUARDIAN Operating System Programming Manual or the ENSCRIBE Programming Manual.

CALL UNLOCKMEMORY (<address> , <byte count>) (II only)

See "Advanced Memory Management" section of the GUARDIAN Operating System Programming Manual.

CALL UNLOCKREC (<file number>
, <tag>)

See "File System Procedures" section of the ENSCRIBE Programming Manual.

CALL USERIDTOUSERNAME (<id name>)

See "Security System" section of the GUARDIAN Operating System Programming Manual.

CALL USERNAMETOUSERID (<name id>)

See "Security System" section of the GUARDIAN Operating System Programming Manual.

→

```
<old segment id> := USESEGMENT ( <segment id> , <pin> )
                                                    (II only)
```

See "Memory Management Procedures" section of the GUARDIAN Operating System Programming Manual.

```
CALL VERIFYUSER ( <user name or id>
                , <logon> , <default> , <default length> )
```

See "Security System" section of the GUARDIAN Operating System Programming Manual, or see the EXPAND Users Manual.

```
<error> := WAIT^FILE ( <file fcb> , <count read> , <time limit> )
```

See "Sequential I/O Procedures" section of the GUARDIAN Operating System Programming Manual.

```
CALL WRITE ( <file number>
           , <buffer>
           , <write count>
           , <count written>
           , <tag>    )
```

See "File System Procedures" section of the GUARDIAN Operating System Programming Manual, the ENSCRIBE Programming Manual, the ENVOY Byte-Oriented Protocols Reference Manual, or the ENVOYACP Bit-Oriented Protocols Reference Manual.

```
{  CALL      } WRITE^FILE ( <file fcb> , <buffer> , <write count>
{ <error> := }              , <reply error code>
                            , <forms control code>
                            , <no wait> )
```

See "Sequential I/O Procedures" section of the GUARDIAN Operating System Programming Manual.

→

```
CALL WRITEREAD ( <file number>
               , <buffer>
               , <write count>
               , <read count>
               , <count read>
               , <tag>   )
```

See "File System Procedures" section of the GUARDIAN Operating System Programming Manual, the ENVOY Byte-Oriented Protocols Reference Manual, or the ENVOYACP Bit-Oriented Protocols Reference Manual.

```
CALL WRITEUPDATE ( <file number>
                 , <buffer>
                 , <write count>
                 , <count written>
                 , <tag>   )
```

See "File System Procedures" section of the GUARDIAN Operating System Programming Manual or the ENSCRIBE Programming Manual.

```
CALL WRITEUPDATEUNLOCK ( <file number>
                       , <buffer>
                       , <write count>
                       , <count written>
                       , <tag>   )
```

See "File System Procedures" section of the ENSCRIBE Programming Manual.

APPENDIX B

FILE SYSTEM ERROR SUMMARY

The table on the following pages provides a quick reference to GUARDIAN file system system errors. For each error, the following information is given:

1. Error number in decimal

2. Error number in octal (in parentheses; the % symbol denotes an octal number)

3. Brief explanation of meaning

4. Device types (kinds of files) for which the errors may occur, as returned by the DEVICEINFO procedure.

The device type numbers in the table correspond to device types as follows:

device type		
0	=	Process (process ID)
1	=	Operator console ($0)
2	=	$RECEIVE
3	=	Disc
4	=	Magnetic tape
5	=	Line printer
6	=	Terminal: conversational or page mode (AXCESS ITI protocol)
7	=	ENVOY data communication line
8	=	Card reader
9	=	AXCESS process-to-process interface (X25AM)
10	=	AXCESS 3271 CRT mode interface (AM3270, TR3271)
11	=	ENVOYACP data communication line
12	=	Tandem to IBM Link (TIL)
20-23	=	Transaction Monitoring Facility (TMF)
26	=	Tandem HyperLink (THL)
59	=	AXCESS data communication line (AM6520)
60	=	AXCESS data communication line (AM3270, TR3271)
61	=	AXCESS data communication line (X25AM)

62 = EXPAND Network Control Process (NCP)
63 = EXPAND line handler

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

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

For more complete information on these errors, including state of the system and suggested corrective action, refer to section 2.4, "File System Errors".

FILE SYSTEM ERROR SUMMARY		
Error Number	Description	Device Type
CONDITION CODE = (CCE): NO ERROR		
0	operation successful	any
CONDITION CODE > (CCG): WARNINGS		
1	end-of-file or end of medium	3,4,6,8
2	operation not allowed on this type file	any
3	failure to open or purge a partition	3
4	failure to open an alternate key file	3
5	failure to provide sequential buffering (I only)	3
6	system message received	2
7	process not accepting CONTROL, SETMODE, or RESETSYNC messages	0
8 (%10)	operation successful (examine MCW for additional status)	11.40,11.42
CONDITION CODE < (CCL): ERRORS		
10 (%12)	file or record already exists	3
11 (%13)	file not in directory or record not in file	3
12 (%14)	file in use	any except 2
13 (%15)	illegal filename specification	any
14 (%16)	device does not exist	any except 1 and 2
15 (%17)	RENAME attempted to another volume	3

FILE SYSTEM ERROR SUMMARY (cont'd)

Error Number	Description	Device Type
16 (%20)	file number has not been opened	any
17 (%21)	attempted CHECKOPEN when file not open by primary, CHECKOPEN parameters do not match those of primary open, or primary process not alive	any
18 (%22)	referenced system does not exist	any
19 (%23)	no more devices in logical device table	any except 1 and 2
20 (%24)	attempted network access by process with five-character name or seven-character home terminal name	any except 2
21 (%25)	illegal count specified	any except 2
22 (%26)	application parameter or buffer address out of bounds	any
23 (%27)	disc address out of bounds	3
24 (%30)	privileged mode required for this operation	any
25 (%31)	AWAITIO or CANCEL attempted on wait file	any
26 (%32)	AWAITIO, CANCEL, or CONTROL 22 attempted on a file with no outstanding requests	any
27 (%33)	wait operation attempted when outstanding requests pending	any
28 (%34)	number of outstanding no-wait operations exceed OPEN specification, or attempt to open disc file or $RECEIVE with maximum number of concurrent operations greater than 1	any
29 (%35)	missing parameter	any

FILE SYSTEM ERROR SUMMARY (cont'd)		
Error Number	Description	Device Type
30 (%36)	unable to obtain main memory space for a link control block	any except 2
31 (%37)	unable to obtain SHORTPOOL space for a file system buffer area (I only); unable to obtain file system buffer space (II only)	any
32 (%40)	unable to obtain main memory space for a control block (I only); unable to obtain storage pool space (SYSPOOL) (II only); or INFO procedure called with <file number> = -1 but no file was open (both systems)	any
33 (%41)	i/o process unable to obtain IOPOOL space for i/o buffer, or count too large for dedicated i/o buffer (I only); i/o process unable to obtain i/o segment space (II only)	any except 2
	read from unstructured disc spans too many sectors	3
34 (%42)	unable to obtain file system control block (II only)	any
35 (%43)	unable to obtain i/o process control block (II only)	any except 2
36 (%44)	unable to obtain physical memory (II only)	any
37 (%45)	unable to obtain physical memory for i/o (II only)	any except 2
38 (%46)	operation attempted on wrong type of system	any except 2
40 (%50)	operation timed out	any
41 (%51)	checksum error on file synchronization block	3

FILE SYSTEM ERROR SUMMARY (cont'd)

Error Number	Description	Device Type
42 (%52)	attempt to read from unallocated extent	3
43 (%53)	unable to obtain disc space for extent	3
44 (%54)	directory is full	3
45 (%55)	file is full	3
46 (%56)	invalid key specified	3
47 (%57)	key not consistent with file data	3
48 (%60)	security violation, or remote password illegal or does not exist	3
49 (%61)	access violation	any except 2
50 (%62)	directory error	3
51 (%63)	directory is bad	3
52 (%64)	error in disc free space table	3
53 (%65)	file system internal error	3
54 (%66)	i/o error in disc free space table	3
55 (%67)	i/o error in directory	3
56 (%70)	i/o error on volume label	3
57 (%71)	disc free space table is full	3
58 (%72)	disc free space table is bad	3
59 (%73)	file is bad	3
60 (%74)	volume on which this file resides has been removed, device has been downed, or process has failed since the file was opened	any except 1 and 2

FILE SYSTEM ERROR SUMMARY (cont'd)

Error Number	Description	Device Type
61 (%75)	no more file opens permitted on this volume	3
62 (%76)	volume has been mounted, but mount order has not been given	3
63 (%77)	volume has been mounted and mount is in progress (waiting for mount interrupt)	3
64 (%100)	volume has been mounted and mount is in progress	3
65 (%101)	only special requests permitted	3
66 (%102)	device has been downed by operator, or hard failure occurred on controller	any except 2
70 (%106)	continue file operation	0, 3
71 (%107)	duplicate record	3
72 (%110)	access to secondary partition not permitted	3
73 (%111)	file or record locked	3
74 (%112)	READUPDATE called for $RECEIVE and number of messages queued exceeds receive depth, REPLY called with an invalid message tag, or REPLY called with no message outstanding	2
75 (%113)	requesting process has no current process TRANSID	3
76 (%114)	transaction is in the process of ending	3 or none
77 (%115)	a TMF system file has the wrong file code	3
78 (%116)	TRANSID is invalid or obsolete	3 or none
79 (%117)	attempt made by TRANSID to update or delete a record it has not previously locked	3

FILE SYSTEM ERROR SUMMARY (cont'd)

Error Number	Description	Device Type
80 (%120)	invalid operation attempted on audited file or non-audited disc volume	3
81 (%121)	attempted operation invalid for TRANSID that has no-wait i/o outstanding on a disc or process file	2 or none
82 (%122)	TMF is not running	0, 3, or none
83 (%123)	process has initiated more concurrent transactions than can be handled	none
84 (%124)	TMF is not configured	0, 3, or none
87 (%127)	waiting on a READ request and did not get it	10
88 (%130)	a CONTROL READ is pending; new READ invalid	10
89 (%131)	remote device has no buffer available	10
90 (%132)	TRANSID aborted because its parent process died	3 or none
91 (%133)	Internal software error	4.2
92 (%134)	TRANSID aborted because path to remote node is down	3 or none
93 (%135)	TRANSID aborted because it spanned too many audit files	3 or none
94 (%136)	TRANSID aborted by operator command	3 or none
97 (%141)	TRANSID was aborted	3 or none
98 (%142)	Transaction Monitor Process's Network Active Transactions table is full	0, 3, or none

FILE SYSTEM ERROR SUMMARY (cont'd)

Error Number	Description	Device Type
99 (%143)	attempt to use microcode option that is not installed	any except 2
100 (%144)	device not ready	any except 2
101 (%145)	no write ring	4
102 (%146)	paper out or bail not properly closed	5
103 (%147)	disc not ready due to power failure	3
104 (%150)	no response from device	5.4
105 (%151)	VFU error	5.4
110 (%156)	only break access permitted	6, 61
111 (%157)	operation aborted because of break	6, 61
112 (%160)	READ or WRITEREAD preempted by operator message	6
	too many user console messages	1
120 (%170)	data parity error	any except 2
121 (%171)	data overrun error	any except 2
122 (%172)	request aborted due to possible data loss caused by reset of circuit	6, 9, 11, 61
123 (%173)	subdevice busy	5, 6, 10
124 (%174)	line reset is in progress	6, 10, 59, 60
130 (%202)	illegal address to disc	3

FILE SYSTEM ERROR SUMMARY (cont'd)

Error Number	Description	Device Type
131 (%203)	write check error from disc	3.0, 3.1
132 (%204)	seek incomplete from disc	3.0, 3.1
133 (%205)	access not ready on disc	3.0, 3.1
134 (%206)	address compare error on disc	3
135 (%207)	write protect violation with disc	3
136 (%210)	unit ownership error (dual-port disc)	3
137 (%211)	controller buffer parity error	any except 2
138 (%212)	interrupt overrun	any except 2
139 (%213)	controller error	any except 2
140 (%214)	modem error, or modem or link disconnected	6, 7, 10, 11, 12, 59, 60, 61, 63
145 (%221)	card reader motion check error	8
146 (%222)	card reader read check error	8
147 (%223)	invalid Hollerith code read	8
150 (%226)	end-of-tape marker detected	4
151 (%227)	runaway tape detected	4
152 (%230)	unusual end -- unit went offline	4, 11
153 (%231)	tape drive power on	4
154 (%232)	BOT detected during backspace files or backspace records	4
155 (%233)	only nine-track tape permitted	4

FILE SYSTEM ERROR SUMMARY (cont'd)		
Error Number	Description	Device Type
156 (%234)	TIL protocol violation detected	12
157 (%235)	i/o process internal error	any except 2
158 (%236)	invalid function requested for HyperLink	26
160 (%240)	request is invalid for line state	6, 7, 10, 11
	more than 7 reads or 7 writes issued	11
161 (%241)	impossible event occurred for line state	7, 10, 11
162 (%242)	operation timed out	7, 10, 11
163 (%243)	EOT received	7.0-7.3,7.8
	power at auto-call unit is off	7.56, 11
164 (%244)	disconnect received	7.0, 7.1, 10, 11, 61
	data line is occupied (busy)	7.56, 11
165 (%245)	RVI received	7.0 - 7.3
	data line not occupied after setting call request	7.56, 11
166 (%246)	ENQ received	7.0, 7.1, 7.3, 7.9
	auto-call unit failed to set "present next digit"	7.56, 11
167 (%247)	EOT received on line bid/select	7.0, 7.1, 7.3, 7.8
	"data set status" not set after dialing all digits	7.56, 11

FILE SYSTEM ERROR SUMMARY (cont'd)

Error Number	Description	Device Type
168 (%250)	NAK received on line bid/select	7.0, 7.1, 7.3, 7.8
	auto-call unit failed to clear "present next digit" after "digit present" was set	7.56, 11
169 (%251)	WACK received on line bid/select	7.0,7.1,7.3
	auto-call unit set "abandon call and retry"	7.56, 11
	station disabled or station not defined	11
170 (%252)	no ID sequence received during circuit assurance mode	7.0, 7.1
	invalid MCW entry number on WRITE	11.40
171 (%253)	no response received on bid/poll/select	7, 10, 11, 61
172 (%254)	reply not proper for protocol	6, 7, 10, 11
173 (%255)	maximum allowable NAKs received (transmission error)	6, 7, 10
	invalid MCW on WRITE	11
174 (%256)	WACK received after select	7.2, 7.3
	aborted transmitted frame	11
175 (%257)	incorrect alternating ACK received	7.0 - 7.3
	command reject	11
176 (%260)	poll sequence ended with no responder	7.3, 7.8, 7.9, 11.40
177 (%261)	text overrun	7, 10, 11

FILE SYSTEM ERROR SUMMARY (cont'd)

Error Number	Description	Device Type
178 (%262)	no address list specified	7.2, 7.3, 7.8, 7.9, 11.40, 61
179 (%263)	application buffer is incorrect	10, 61
	control request pending or autopoll active	11.40
180 (%264)	unknown device status received	6.6 - 6.10, 5.3, 5.4, 10
181 (%265)	status receipt currently enabled for subdevice	10
190 (%276)	invalid status received from device	any except 2
191 (%277)	device power on	5
192 (%300)	device is being exercised	3 - 6
193 (%301)	invalid or missing microcode files	4.2
200 (%310)	device is owned by alternate port	any except 2
201 (%311)	current path to the device is down	any except 0 and 2
	attempt was made to write to a nonexistent process	0
210 (%322)	device ownership changed during operation	any except 2
211 (%323)	failure of cpu performing this operation	any
212 (%324)	EIO instruction failure (I only)	any except 2
213 (%325)	channel data parity error	any except 2

FILE SYSTEM ERROR SUMMARY (cont'd)

Error Number	Description	Device Type
214 (%326)	channel timeout	any except 2
215 (%327)	i/o attempted to absent memory page	any except 2
216 (%330)	map parity error during this i/o (I only); or memory access breakpoint during this i/o (II only)	any except 2
217 (%331)	memory parity error during this i/o	any except 2
218 (%332)	interrupt timeout	any except 2
219 (%333)	illegal device reconnection	any except 2
220 (%334)	protect violation	any except 2
221 (%335)	channel pad-in violation (I only); controller handshake violation (II only)	any except 2
222 (%336)	bad channel status from EIO instruction	any except 2
223 (%337)	bad channel status from IIO instruction	any except 2
224 (%340)	controller error (I only)	any except 2
225 (%341)	no unit or multiple units assigned to same unit number	any except 2
226 (%342)	controller busy error (I only)	any except 2

FILE SYSTEM ERROR SUMMARY (cont'd)		
Error Number	Description	Device Type
230 (%346)	cpu power on during this operation	any except 2
231 (%347)	controller power on during this operation	any except 2
240 (%350)	network line handler error; operation not started	any except 2
241 (%351)	network error; operation not started	any except 2
248 (%370)	network line handler process failed while this request was outstanding	any except 2
249 (%371)	network failure occurred while this request was outstanding	any except 2
250 (%372)	all paths to the system are down	any except 2
251 (%373)	network protocol error occurred	any except 2
300-511	reserved for use by application processes	

APPENDIX C

SYSTEM MESSAGES

The following messages from the operating system may be sent to an application process through the $RECEIVE file.

The first word of a system message always has a value less than zero. Also, the completion of a read associated with a system message returns a condition code of CCG (greater than) and error 6 from FILEINFO.

Note: Like all interprocess messages, system messages read via calls to the READUPDATE procedure must be replied to in a corresponding call to REPLY. If the application process is performing message queueing, LASTRECEIVE or RECEIVEINFO must also be called immediately following completion of the READUPDATE, and the message tag must be passed back to the REPLY procedure.

The system messages and their formats, in word elements, are as follows:

- CPU Down Message. There are two forms of the CPU Down message:

```
<sysmsg>            = -2
<sysmsg>[1]         = cpu
```

  This form is received if a failure occurs with a processor module being monitored. Monitoring for specific processor modules is requested by a call to the process control MONITORCPUS procedure.

  and

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

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

- CPU Up Message

```
<sysmsg>            = -3
<sysmsg>[1]         = cpu
```

This message is received when a processor module being monitored is reloaded.

- Process Normal Deletion (STOP) Message

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

There are two forms of the STOP message:

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

This form is received by a deleted process's creator if the deleted process was not named or by one member of a process pair when the other member is deleted.

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

This form is received by a process pair's ancestor when the process name is deleted from the PPD. This indicates that neither member of the process pair exists.

- Process Abnormal Deletion (ABEND) Message

This message is received if the deletion is due to a call to the process control ABEND procedure or because the deleted process encountered a trap condition and was aborted by the operating system.

There are two forms of the ABEND message:

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

This form is received by a deleted process's creator if the deleted process was not named or by one member of a process pair when the other member is deleted.

```
<sysmsg>            = -6
<sysmsg>[1] FOR 3   = $<process name> of deleted process [pair]
<sysmsg>[4]         = -1
```

This form is received by a process pair's ancestor when the process name is deleted from the PPD. This indicates that neither member of the process pair exists.

- Change in Status of Network Nodes

```
<sysmsg>              = -8
<sysmsg>[1].<0:7>     = system number
<sysmsg>[1].<8:15>    = number of cpu's
<sysmsg>[2]           = current processor status bitmask
<sysmsg>[3]           = previous processor status bitmask
```

This message is received if the process is running on a system that is part of a network, and has enabled receipt of remote status change messages by passing "1" as a parameter to the MONITORNET procedure.

- SETTIME Message (NonStop II systems only)

```
<sysmsg>              = -10
<sysmsg>[1]           = cpu
```

This message is received if the interval clock of "cpu" has been reset by the system manager or operator, provided the process has enabled receipt of new messages by a call to MONITORNEW.

- Power On Message (NonStop II systems only)

```
<sysmsg>              = -11
<sysmsg>[1]           = cpu
```

This message is received if the indicated processor had a POWER OFF, then a POWER ON condition, provided the process has enabled receipt of new messages by a call to MONITORNEW.

- NEWPROCESSNOWAIT Completion Message (NonStop II systems only)

```
<sysmsg>              = -12
<sysmsg>[1]           = error
<sysmsg>[2] FOR 2     = tag
<sysmsg>[4] FOR 4     = process ID
```

This message is received by a process when a call to the NEWPROCESSNOWAIT procedure is completed.

- BREAK Received from Terminal

```
<sysmsg>              = - 20
<sysmsg>[1]           = logical device number, in binary, of
                        device where break was typed
<sysmsg>[2]           = system number, in binary, of logical
                        device number
```

This message is received by a process if it has specified break monitoring (through a call to SETMODE or SETMODENOWAIT) and and BREAK is typed on a terminal being monitored.

- Time Signal Message (NonStop II systems only)

```
<sysmsg>              = -22
<sysmsg>[1]           = <parameter1> supplied to SIGNALTIMEOUT
                            (if none supplied, 0)
<sysmsg>[2] FOR 2     = <parameter2> supplied to SIGNALTIMEOUT
                            (if none supplied, 0D)
```

This message is received if a timer set by a call to SIGNALTIMEOUT has timed out.

- Memory Lock Completion Message (NonStop II systems only)

```
<sysmsg>              = -23
<sysmsg>[1]           = <parameter1> supplied to LOCKMEMORY
                            (if none supplied, 0)
<sysmsg>[2] FOR 2     = <parameter2> supplied to LOCKMEMORY
                            (if none supplied, 0D)
```

This message is received if a call to LOCKMEMORY waited for memory, but completed successfully before the specified time limit was reached.

- Memory Lock Failure Message (NonStop II systems only)

```
<sysmsg>              = -24
<sysmsg>[1]           = <parameter1> supplied to LOCKMEMORY
                            (if none supplied, 0)
<sysmsg>[2] FOR 2     = <parameter2> supplied to LOCKMEMORY
                            (if none supplied, 0D)
```

This message is received if a call to LOCKMEMORY waited for memory and timed out without completing the lock.

Receipt of the following five system messages (OPEN, CLOSE, CONTROL, SETMODE, and RESETSYNC) is possible only if the process has opened its $RECEIVE file with <flags>.<1> = 1:

- Process OPEN Message

```
<sysmsg>              = -30
<sysmsg>[1]           = <flags> parameter to caller's OPEN
<sysmsg>[2]           = <sync or receive depth> parameter to
                        caller's OPEN
<sysmsg>[3] FOR 4     = 0 if normal open, process ID of primary
                        process if an open by a backup process
<sysmsg>[7]           = 0 if normal open, file number of file if
                        an open by a backup process
<sysmsg>[8]           = process accessor ID of opener
<sysmsg>[9] FOR 4     = optional 1st qualif name of named
                        process or blanks
<sysmsg>[13] FOR 4    = optional 2nd qualif name of named
                        process or blanks
```

This message is received by a process when it is opened by another process. The process ID of the opener can be obtained in a subsequent call to LASTRECEIVE or RECEIVEINFO.

Note: This message is also received if the open is by the backup process of a process pair. Therefore, a process can expect two of these messages when being opened by a process pair.

- Process CLOSE Message

```
<sysmsg>            = -31
```

This message is received by a process when it is closed by another process. The process ID of the closer can be obtained in a subsequent call to LASTRECEIVE or RECEIVEINFO.

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

- Process CONTROL Message

```
<sysmsg>            = -32
<sysmsg>[1]         = <operation> parameter to caller's CONTROL
<sysmsg>[2]         = <parameter> parameter to caller's CONTROL
```

This message is received when another process calls the CONTROL procedure referencing the receiver process file. The process ID of the caller to CONTROL can be obtained in a subsequent call to LASTRECEIVE or RECEIVEINFO.

- Process SETMODE Message

```
<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
```

This message is received when another process calls the SETMODE or SETMODENOWAIT procedure referencing the receiver process file. The process ID of the caller to SETMODE or SETMODENOWAIT can be obtained in a subsequent call to LASTRECEIVE or RECEIVEINFO.

- Process RESETSYNC Message

```
<sysmsg>            = -34
```

This message is received when a process calls the RESETSYNC procedure referencing the receiver process file (note that a call to the CHECKPOINT procedure may contain an implicit call to RESETSYNC). This means that the sync ID value for that file has

been reset to zero. Therefore, a server process using the sync ID mechanism should clear its local copy of the sync ID value.

The process ID of the caller to RESETSYNC can be obtained in a subsequent call to LASTRECEIVE or RECEIVEINFO.

COMMAND INTERPRETER MESSAGES

The following messages may be received from the Command Interpreter. These are not system messages; i.e., they do not cause an error 6 indication to be returned.

Startup Message

The startup message is sent to the new process immediately following the successful creation of the new process. The startup message is read by the process via its $RECEIVE file.

The form of the startup parameter message is:

```
STRUCT ci^startup;
  BEGIN                          !  word
    INT msgcode;                 !  [0] -1.
    STRUCT default;
      BEGIN
        INT volume [ 0:3 ], !    [1] $<default volume name>.
            subvol [ 0:3 ]; !        <default subvol name>.
      END;
    STRUCT infile;
      BEGIN
        INT volume [ 0:3 ], !    [9] IN parameter <file name> of RUN
            subvol [ 0:3 ], !        command.
            dname  [ 0:3 ];
      END;
    STRUCT outfile;
      BEGIN
        INT volume [ 0:3 ], ! [21] OUT parameter <file name> of RUN
            subvol [ 0:3 ], !        command.
            dname  [ 0:3 ];
      END;
    STRING param [ 0:n-1 ]; ! [33] <parameter string> of RUN
  END; ! ci^startup^msg.     !      Command (if any) that was was
                             !      entered by operator.  This is in
                             !      either of the following forms:
                             !
                             !      <parameter string><null>[<null>]
                             !
                             !      or
                             !
                             !      <null><null>
                             !
                             !      <n> = ( <count read> - 66 )
```

The maximum length possible for a startup message is 594 bytes (including the trailing null characters).

Note: The parameter message length is always an even number. If necessary, the Command Interpreter will pad the parameter string with an additional null.

Assign Message

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

- replies to the startup message with an error return value of REPLY = 70. The Command Interpreter then sends both assign and param messages.

- replies to the startup message with an error return value of 0, but with a reply of one to four bytes, and bit 0 of the first byte of the reply is set to 1. The Command Interpreter also sends param messages if bit 1 of the first byte of the reply is set to 1.

The form of the assign message is:

```
STRUCT ci^assign;                 ! assign message.
  BEGIN                           !
    INT msg^code;                 !  [0] -2
                                  !
    STRUCT logicalunit;           ! PARAMETERS TO ASSIGN COMMAND.
      BEGIN                       !
        STRING prognamelen,       !  [1] length in bytes of name {0:31}
               progname[0:30],    !      { <program unit> | *}<blanks>
               filenamelen,       ! [17] length in bytes of name {0:31}
               filename[0:30];    !      <logical file><blanks>
      END;                        !
    INT(32) fieldmask;            ! [33] bit mask to indicate which of
                                  !      the following fields were
                                  !      supplied (1 = supplied):
                                  !
                                  !      .<0> = <Tandem file name>
                                  !      .<1> = <pri extent size>
                                  !      .<2> = <sec extent size>
                                  !      .<3> = <file code>
                                  !      .<4> = <exclusion spec>
                                  !      .<5> = <access spec>
                                  !      .<6> = <record size>
                                  !      .<7> = <block size>
                                  !
    STRUCT tandemfilename;        ! [35] <Tandem file name>
      BEGIN                       !
        INT volume  [ 0:3 ],      !
            subvol  [ 0:3 ],      !
            dfile   [ 0:3 ];      !
      END;                        !
    ! createspec                 !
    INT primaryextent,            ! [47] <pri extent size>.
        secondaryextent,          ! [48] <sec extent size>.
        filecode,                 ! [49] <file code>.
        exclusionspec,            ! [50]   %00 if SHARED,     } corre-
                                  !        %20 if EXCLUSIVE,  } sponds
                                  !        %60 if PROTECTED.  } to flag
        accessspec,               ! [51] %0000 if I-O,        } param of
                                  !      %2000 if INPUT,      } OPEN.
                                  !      %4000 if OUTPUT.     }
        recordsize,               ! [52] <record size>.
        blocksize;                ! [53] <block size>.
  END;
```

The length of this message is 108 bytes.

Param Message

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

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

The form of the param message is:

```
STRUCT ci^param;                    ! param message.
  BEGIN                             !
    INT msg^code,                   !  [0] -3
        numparams;                  !  [1] number of parameters
                                    !      included in this message.
    STRING parameters [ 0:1023 ]; !  [2] beginning of parameters.
  END;
```

The field "parameters" in the above message format is comprised of "numparams" records of the form (offsets are given in bytes):

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

The maximum length of this message is 1028 bytes.

Wakeup Message

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

If the Command Interpreter is not in the pause state (i.e., it is prompting for a command or executing a command other the RUN), a wakeup message will be ignored.

The form of the wakeup message is:

```
STRUCT wakeup^msg;
  BEGIN
    INT msgcode; ! -20
  END;
```

The length of this message is two (2) bytes.

Display Message

The display message, when received by a Command Interpreter, causes the Command Interpreter to display the text contained in the message. The text is displayed just prior to the next time that the Command Interpreter prompts for a command (i.e., issues a ":").

A Command Interpreter has the capability store one 132-byte display message until it is able to display the message text. If the Command Interpreter is currently storing a display message when another display message is sent to it, the second another display message will be rejected with an error 12 indication (file in use).

The form of the display message is

```
STRUCT display^msg;
  BEGIN
    INT msgcode;            ! -21
    STRING text [ 0:n-1 ]; ! n <= 132.
  END;
```

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

Logon Message

This message is sent to the $CMON process if it exists when a LOGON command is entered and the user name is checked for validity.

The form of the logon message is:

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

The length of this message is 54 bytes.

The form of the reply to the logon message is:

```
STRUCT logon^reply;
  BEGIN
    INT replycode;            !  [0] 0 = allow logon.
                              !      1 = disallow logon.
    STRING                    !
        replytext [ 0:131 ]; !  [1] optional message to be printed.
  END;
```

The length of this message is 2 + reply text length in bytes. Note that the length of the reply text is implied in the reply count used when making a reply. If <reply count> = 2, no text will be displayed.

Logoff Message

This message is sent to the $CMON process if it exists when a LOGOFF command is entered.

The form of the logoff message is:

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

The length of this message is 54 bytes.

The form of the reply to the logoff message is:

```
STRUCT logoff^reply;
  BEGIN
    INT replycode;            !  [0] 0 or 1.
    STRING                    !
        replytext [ 0:131 ]; !  [1] optional message to be printed
  END;
```

The length of this message is 2 + reply text length in bytes. Note that the length of the reply text is implied in the reply count used when making a reply. If <reply count> = 2, no text will be displayed.

Process Creation Message

This message is sent to the $CMON process if it exists when an implicit or explicit RUN command is entered.

The form of this message is:

```
STRUCT processcreation^msg;
  BEGIN
    INT msgcode,              !  [0] -52.
        userid,               !  [1] user ID of user logging on
        cipri,                !  [2] execution priority if CI
        ciinfile [ 0:11 ],    !  [3] name of CI's command file
        cioutfile[ 0:11 ],    ! [15] name of CI's list file
        progname [ 0:11 ],    ! [27] expanded program file name
        priority,             ! [39] <priority> of RUN command if
                              !      supplied; otherwise -1
        processor;            ! [40] <processor module> of RUN
                              !      command if supplied; otherwise
                              !      -1
  END;
```

The length of this message is 82 bytes.

The $CMON process may reply in either of two ways. The first causes a process creation to be attempted. This form of reply is:

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

The values returned in this reply are those used for the process creation attempt. Any process creation errors will be seen by the Command Interpreter user (no notification will be made to $CMON).

The second form of reply is used to disallow the process creation. This form of reply is:

```
STRUCT processcreation^reply;
  BEGIN
    INT replycode;            !  [0] 1 = disallow process creation.
    STRING                    !
        replytext [ 0:131 ]; !  [1] optional message to be printed.
  END;
```

The length of this message is 2 + reply text length in bytes. Note that the length of the reply text is implied in the reply count used when making a reply. If <reply count> = 2, no text will be displayed.

APPENDIX D

SOURCE FOR $SYSTEM.SYSTEM.GPLDEFS

The following is the $SYSTEM.SYSTEM.GPLDEFS source file for the sequential i/o procedures described in section 9.

```
?PAGE "T9600D00 - SIO PROCEDURES - DEFINITIONS"
!
! FCB SIZE IN WORDS.
!
LITERAL
     FCBSIZE = 60;
!
! DECLARE RUCB , PUCB, AND COMMON FCB.
!
DEFINE
     ALLOCATE^CBS ( RUCB^NAME , COMMON^FCB^NAME , NUM^FILES ) =
        INT .RUCB^NAME [ 0:65 ] :=
               ! RUCB PART.
               [ 62 , 1 , 27 * [ 0 ] , 62 , 32 * [ 0 ] ,
               ! PUCB PART.
                 4 , NUM^FILES , 4 + FCBSIZE ];
        INT .COMMON^FCB^NAME [ 0:FCBSIZE - 1 ] :=
               [ FCBSIZE * [ 0 ] ]#;
!
! DECLARE FCB.
!
DEFINE
     ALLOCATE^FCB ( FCB^NAME , PHYS^FILENAME ) =
        INT .FCB^NAME [ 0:FCBSIZE - 1 ] :=
               [ FCBSIZE , %000061 , -1 , %100000 , 0 , PHYS^FILENAME,
                 ( FCBSIZE - 17 ) * [ 0 ] ]#;
!
! OPEN ACCESS.
!
LITERAL
     READWRITE^ACCESS = 0,
     READ^ACCESS      = 1,
     WRITE^ACCESS     = 2;
!
! OPEN EXCLUSION.
!
```

```
LITERAL
     SHARE     = 0,
     EXCLUSIVE = 1,
     PROTECTED = 3;
!
! OPEN^FILE FLAGS                    V         111111
!                                    0123456789012345
                        !      1111111111222222222233
DEFINE                  !4567890123456789012345678901
     ABORT^OPENERR = %B00000000000000000000000000001D#,
     ABORT^XFERERR = %B00000000000000000000000000010D#,
     PRINT^ERR^MSG = %B00000000000000000000000000100D#,
     AUTO^CREATE   = %B00000000000000000000000001000D#,
     MUSTBENEW     = %B00000000000000000000000010000D#,
     PURGE^DATA    = %B00000000000000000000000100000D#,
     AUTO^TOF      = %B00000000000000000000001000000D#,
     NOWAIT        = %B00000000000000000000010000000D#,
     BLOCKED       = %B00000000000000000000100000000D#,
     VAR^FORMAT    = %B00000000000000000001000000000D#,
     READ^TRIM     = %B00000000000000000010000000000D#,
     WRITE^TRIM    = %B00000000000000000100000000000D#,
     WRITE^FOLD    = %B00000000000000001000000000000D#,
     WRITE^PAD     = %B00000000000000010000000000000D#,
     CRLF^BREAK    = %B00000000000000100000000000000D#;
!
! SET^FILE OPERATIONS.
!
LITERAL
      INIT^FILEFCB                =   0,
      !
      ASSIGN^FILENAME             =   1,
      ASSIGN^LOGICALFILENAME      =   2,
      ASSIGN^OPENACCESS           =   3,
      ASSIGN^OPENEXCLUSION        =   4,
      ASSIGN^RECORDLENGTH         =   5,
      ASSIGN^RECORDLEN            =   ASSIGN^RECORDLENGTH,
      ASSIGN^FILECODE             =   6,
      ASSIGN^PRIMARYEXTENTSIZE    =   7,
      ASSIGN^PRIEXT               =   ASSIGN^PRIMARYEXTENTSIZE,
      ASSIGN^SECONDARYEXTENTSIZE  =   8,
      ASSIGN^SECEXT               =   ASSIGN^SECONDARYEXTENTSIZE,
      ASSIGN^BLOCKLENGTH          =   9,
      ASSIGN^BLOCKBUFLEN          =   ASSIGN^BLOCKLENGTH,
      !
      SET^DUPFILE                 =  10,
      SET^SYSTEMMESSAGES          =  11,
      SET^OPENERSPID              =  12,
      SET^RCVUSEROPENREPLY        =  13,
      SET^RCVOPENCNT              =  14,
      SET^RCVEOF                  =  15,
      SET^USERFLAG                =  16,
      SET^ABORT^XFERERR           =  17,
      SET^PRINT^ERR^MSG           =  18,
      SET^READ^TRIM               =  19,
```

```
SET^WRITE^TRIM                = 20,
SET^WRITE^FOLD                = 21,
SET^WRITE^PAD                 = 22,
SET^CRLF^BREAK                = 23,
SET^PROMPT                    = 24,
SET^ERRORFILE                 = 25,
SET^PHYSIOOUT                 = 26,
SET^LOGIOOUT                  = 27,
SET^COUNTXFERRED              = 28,
SET^ERROR                     = 29,
SET^BREAKHIT                  = 30,
SET^TRACEBACK                 = 31,
!
SET^EDITREAD^REPOSITION       = 32,
!
FILE^FILENAME^ADDR            = 33,
FILE^LOGICALFILENAME^ADDR     = 34,
FILE^FNUM^ADDR                = 35,
FILE^ERROR^ADDR               = 36,
FILE^USERFLAG^ADDR            = 37,
FILE^SEQNUM^ADDR              = 38,
FILE^FILEINFO                 = 39,
FILE^CREATED                  = 40,
FILE^FNUM                     = 41,
FILE^SEQNUM                   = 42,
FILE^ASSIGNMASK1              = 43,
FILE^ASSIGNMASK2              = 44,
FILE^FWDLINKFCB               = 45,
FILE^BWDLINKFCB               = 46,
!
SET^CHECKSUM                  = 47,
!
FILE^OPENERSPID^ADDR          = 48,
!
SET^SYSTEMMESSAGESMANY        = 49,
!
MAX^OPERATION                 = 49,
!
FILE^FILENAME                 = ASSIGN^FILENAME             + 256,
FILE^LOGICALFILENAME          = ASSIGN^LOGICALFILENAME      + 256,
FILE^OPENACCESS               = ASSIGN^OPENACCESS           + 256,
FILE^OPENEXCLUSION            = ASSIGN^OPENEXCLUSION        + 256,
FILE^RECORDLEN                = ASSIGN^RECORDLENGTH         + 256,
FILE^FILECODE                 = ASSIGN^FILECODE             + 256,
FILE^PRIEXT                   = ASSIGN^PRIMARYEXTENTSIZE    + 256,
FILE^SECEXT                   = ASSIGN^SECONDARYEXTENTSIZE  + 256,
FILE^BLOCKBUFLEN              = ASSIGN^BLOCKLENGTH          + 256,
FILE^DUPFILE                  = SET^DUPFILE                 + 256,
FILE^SYSTEMMESSAGES           = SET^SYSTEMMESSAGES          + 256,
FILE^OPENERSPID               = SET^OPENERSPID              + 256,
FILE^RCVUSEROPENREPLY         = SET^RCVUSEROPENREPLY        + 256,
FILE^RCVOPENCNT               = SET^RCVOPENCNT              + 256,
FILE^RCVEOF                   = SET^RCVEOF                  + 256,
FILE^USERFLAG                 = SET^USERFLAG                + 256,
```

```
      FILE^ABORT^XFERERR          = SET^ABORT^XFERERR          + 256,
      FILE^PRINT^ERR^MSG          = SET^PRINT^ERR^MSG          + 256,
      FILE^READ^TRIM              = SET^READ^TRIM              + 256,
      FILE^WRITE^TRIM             = SET^WRITE^TRIM             + 256,
      FILE^WRITE^FOLD             = SET^WRITE^FOLD             + 256,
      FILE^WRITE^PAD              = SET^WRITE^PAD              + 256,
      FILE^CRLF^BREAK             = SET^CRLF^BREAK             + 256,
      FILE^PROMPT                 = SET^PROMPT                 + 256,
      FILE^ERRORFILE              = SET^ERRORFILE              + 256,
      FILE^PHYSIOOUT              = SET^PHYSIOOUT              + 256,
      FILE^LOGIOOUT               = SET^LOGIOOUT               + 256,
      FILE^COUNTXFERRED           = SET^COUNTXFERRED           + 256,
      FILE^ERROR                  = SET^ERROR                  + 256,
      FILE^BREAKHIT               = SET^BREAKHIT               + 256,
      FILE^TRACEBACK              = SET^TRACEBACK              + 256,
      FILE^CHECKSUM               = SET^CHECKSUM               + 256,
      FILE^SYSTEMMESSAGESMANY     = SET^SYSTEMMESSAGESMANY     + 256;
!
! SIO PROCEDURE ERRORS.
!
LITERAL
     SIOERR^INVALIDPARAM      = 512, ! parameter is invalid.
     SIOERR^MISSINGFILENAME   = 513, ! file name not supplied.
     SIOERR^DEVNOTSUPPORTED   = 514, ! device not supported.
     SIOERR^INVALIDACCESS     = 515, ! access mode incompatible with
                                     ! device.
     SIOERR^INVALIDBUFADDR    = 516, ! buffer address not in lower 32k.
     SIOERR^INVALIDFILECODE   = 517, ! file code of file does not match
                                     ! assigned file code.
     SIOERR^BUFTOOSMALL       = 518, ! buffer too small for edit write
                                     ! (i.e., less than 1024 bytes) or
                                     ! buffer not sufficient for record
                                     ! length.
     SIOERR^INVALIDBLKLENGTH  = 519, ! assign block length > block
                                     ! buffer length.
     SIOERR^INVALIDRECLENGTH  = 520, ! record length = 0, record
                                     ! length > maxrecordlength of
                                     ! OPEN^FILE, record length for
                                     ! $RECEIVE file < 14, or record
                                     ! length > 254 and variable
                                     ! records specified.
     SIOERR^INVALIDEDITFILE   = 521, ! edit file is invalid.
     SIOERR^FILEALREADYOPEN   = 522, ! OPEN^FILE called for file
                                     ! already open.
     SIOERR^EDITREADERR       = 523, ! edit read error.
     SIOERR^FILENOTOPEN       = 524, ! file not open.
     SIOERR^ACCESSVIOLATION   = 525, ! access not in effect for
                                     ! requested operation.
     SIOERR^NOSTACKSPACE      = 526, ! insufficient stack space for
                                     ! temporary buffer allocation.
     SIOERR^BLOCKINGREQD      = 527, ! block buffer required for
                                     ! no-wait fold or pad.
     SIOERR^EDITDIROVERFLOW   = 528, ! edit write directory overflow.
     SIOERR^INVALIDEDITWRITE  = 529, ! write attempted after directory
```

```
                                    ! has been written.
SIOERR^INVALIDRECVWRITE = 530,      ! write to $RECEIVE does not
                                    ! follow read.
SIOERR^CANTOPENRECV     = 531,      ! can't open $RECEIVE for break
                                    ! monitoring.
SIOERR^IORESTARTED      = 532,      ! no-wait i/o restarted.
SIOERR^INTERNAL         = 533,      ! internal error.
SIOERR^CHECKSUMCOMM     = 534,      ! common FCB checksum error.
SIOERR^CHECKSUM         = 535;      ! file FCB checksum error.
```

APPENDIX E

SEQUENTIAL I/O FILE CONTROL BLOCK FORMAT

The following is the internal structure of the File Control Block (FCB) for the sequential i/o procedures described in section 9.

Note: The FCB is included as a debugging aid only. Tandem Computers, Incorporated, reserves the right to make changes to the FCB structure. Therefore, this information must not be used to make program references to elements within the File Control Block.

```
!
! File Control Block (FCB) Structure Template.
!
STRUCT FCB^TMPL ( * );
  BEGIN
    INT SIZE,                    ! ( 0) size of FCB in words.
        NAMEOFFSET,              ! ( 1) word offset to name.
        FNUM;                    ! ( 2) GUARDIAN file number,
    !                                   -1 = closed.
    !
    ! create/open options group.
    !
    INT OPTIONS1,                ! ( 3) assign options.
        OPTIONS2,                ! ( 4) assign options.
        FILENAME [ 0:11 ],       ! ( 5) Tandem file name.
        !
        ! create options.
        !
        FCODE,                   ! (17) file code.
        PRIEXT,                  ! (18) primary extent size in pages.
        SECEXT,                  ! (19) secondary extent size in
        !                               pages.
        RECLEN,                  ! (20) logical record length.
        BLKBUFLEN,               ! (21) block length from ASSIGN,
        !                               block buffer length
        !                               following OPEN^FILE.
        !
        ! open options.
        !
        OPENEXCLUSION,           ! (22) exclusion bits to OPEN.
```

```
        OPENACCESS;             ! (23) access bits to OPEN.
    !
    ! initializer group.
    !
    INT PUCB^POINTER,           ! (24) not used by SIO procedures.
        SAMEFILELINK;           ! (25) not used by SIO procedures.
    !
    ! beginning of sio groups.
    !
    INT FWDLINK,                ! (26) forward link.
        BWDLINK,                ! (27) backward link.
        ADDR,                   ! (28) address of this FCB.
        COMMONFCBADDR,          ! (29) address of common FCB.
        ERROR;                  ! (30) last error.
    !
    ! file FCB section.
    !
    INT DEVINFO,                ! (31) file type, dev type, dev
                                !      subtype.
        OPENFLAGS1,             ! (32) access mode, flags parameters
                                !      to OPEN^F&ILE.
        OPENFLAGS2,             ! (33) flags parameters to OPEN^FILE.
        XFERCNTL1,              ! (34) iotype, sysbuflen, interactive
                                !      prompt.
        XFERCNTL2,              ! (35) physioout, logioout, write
                                !      flush, retry count, edit write
                                !      control.
        DUPFCBADDR;             ! (36) FCB address of file where data
                                !      read from this file is to be
                                !      written.
    INT(32)
        LINENO;                 ! (37) line number from edit read or
                                !      ordinal record count scaled by
                                !      1000.
        !
        ! Data Transfer/Blocking Group.
        !
    INT BLKBUFADDR,             ! (39) word address of block buffer.
        BLKXFERCNT,             ! (40) number of bytes to be
                                !      transferred between device and
                                !      target buffer.
        BLKREADCNT =            ! (40) number of bytes to be read
          BLKXFERCNT,           !      from device to target buffer.
        BLKWRITECNT =           ! (40) number of bytes to be written
          BLKXFERCNT,           !      from target buffer to device.
        BLKCNTXFERRED,          ! (41) number of bytes transferred
                                !      between device and target
                                !      buffer.
        BLKCNTREAD =            ! (41) number of bytes read into
          BLKCNTXFERRED,        !      target buffer.
        BLKCNTWRITTEN =         ! (41) number of bytes written from
          BLKCNTXFERRED,        !      target buffer.
        BLKNEXTREC,             ! (42) (byte address) While blocking/
                                !      deblocking this is the address
```

```
                                !       of the next record pointer in
                                !       the block buffer.
    USRBUFADDR,                 ! (43) byte address of user buffer.
    USRWRCNT,                   ! (44) <write count> parameter of
                                !       WRITE^FILE, <prompt count>
                                !       parameter of READ^FILE.
    USRRDCNT,                   ! (45) <max read count> parameter of
                                !       READ^FILE.
    TFOLDLEN,                   ! (46) terminal write fold length
                                !       (= physical record length).
    USRCNTRD =                  ! (46) number of bytes read into user
      TFOLDLEN,                 !       buffer.
    PHYSXFERCNT,                ! (47) transfer count value passed to
                                !       file system in SIO^PIO.
    PHYSIOCNTXFERRED,           ! (48) count transferred value
                                !       returned from file system
                                !       procedure.
    PHYSIOCNTRD =               ! (48) count read value returned from
       PHYSIOCNTXFERRED,        !       file system.
    PHYSIOCNTWR =               ! (48) count written value returned
       PHYSIOCNTXFERRED;        !       from file system.
!
INT USERFLAG =                  ! (24) flag word to be set by user.
    PUCB^POINTER;
!
! initializer group.
!
INT LOGICALFILENAME [ 0:3 ]; ! (49) logical file name of this file
                                !       to INITIALIZER.
!
! common FCB section.
!
    !
    ! Break Group.
    !
INT BRKFCBADDR =                ! (31) FCB of file owning BREAK.
      DEVINFO,
    BRKMSG =                    ! (32:33) BREAK message buffer.
      OPENFLAGS1,
    BRKCNTL =                   ! (34) break control.
      XFERCNTL1,
    BRKLASTOWNER =              ! (35) BREAK last owner.
      XFERCNTL2;
    !
    ! $RECEIVE Group.
    !
    ! DUPFCBADDR skipped; was system messages mask.
    !
INT RCVCNTL =                   ! (37) $RECEIVE control.
      LINENO,
    PRIMARYPID [-1:-1] =        ! (38:41) Primary opener's
      LINENO,                   !         <process id>.
    BACKUPPID =                 ! (42:45) Backup opener's
      BLKNEXTREC,               !         <process id>.
```

```
        REPLYCODE =                 ! (46) $RECEIVE reply error code.
          TFOLDLEN;
        !
        ! Misc Group.
        !
    INT COMMCNTL =                  ! (47)
          PHYSXFERCNT,
        OPRQSTFCBADDR =             ! (48) FCB of file for which operator
          PHYSIOCNTXFERRED,         !      console messages are being
                                    !      displayed. (see NO^ERROR,
                                    !      prompt).
        OPRQSTCOUNT =               ! (49) Count of number of operator
          LOGICALFILENAME,          !      messages displayed.  (see
                                    !      NO^ERROR, prompt).
        ERRFCBADDR [-1:-1] =        ! (50) FCB address of file where
          LOGICALFILENAME;          !      errors are to be reported.
    !
    INT PXCNT,                      ! (53) Length of partial record
                                    !      transferred between user
                                    !      buffer and block buffer.
        PRCNT = PXCNT,              !      Partial read record length.
        PWCNT = PXCNT;              !      Partial write record length.
    !
    INT SYSMSGS1,                   ! (54) System messages to be
        SYSMSGS2,                   ! (55) passed back to caller.
        SYSMSGS3,                   ! (56)
        SYSMSGS4;                   ! (57)
    !
    INT SPARE1;                     ! (58) Unused FCB word.
    !
    INT CHECKSUM;                   ! (59) Checksum.  If <> 0, check.
    !
  END; ! FCB^TMPL.
!
! -- BIT FIELDS.
!
! - ASSIGN BITS.
!
DEFINE
    !
    FILENAMESUPPLD  = <0>#,
      FCB^FILENAMESUPPLD      = FCB.OPTIONS1.FILENAMESUPPLD#,
    !
    PRIEXTSUPPLD    = <1>#,
      FCB^PRIEXTSUPPLD        = FCB.OPTIONS1.PRIEXTSUPPLD#,
    !
    SECEXTSUPPLD    = <2>#,
      FCB^SECEXTSUPPLD        = FCB.OPTIONS1.SECEXTSUPPLD#,
    !
    FCODESUPPLD     = <3>#,
      FCB^FCODESUPPLD         = FCB.OPTIONS1.FCODESUPPLD#,
    !
    EXCLUSIONSUPPLD = <4>#,
      FCB^EXCLUSIONSUPPLD     = FCB.OPTIONS1.EXCLUSIONSUPPLD#,
```

```
    !
    ACCESSSUPPLD    = <5>#,
      FCB^ACCESSSUPPLD          = FCB.OPTIONS1.ACCESSSUPPLD#,
    !
    RRECLENSUPPLD   = <6>#,
      FCB^RRECLENSUPPLD         = FCB.OPTIONS1.RRECLENSUPPLD#,
    !
    BLOCKLENSUPPLD  = <7>#,
      FCB^BLOCKLENSUPPLD        = FCB.OPTIONS1.BLOCKLENSUPPLD#;
!
! - OPEN EXCLUSION (FCB^OPENEXCLUSION)
!
DEFINE
     EXCLUSIONFIELD = <9:11>#,
       FCB^EXCLUSIONFIELD       = FCB.OPENEXCLUSION.EXCLUSIONFIELD#;
!
! - OPEN ACCESS (FCB^OPENACCESS)
!
DEFINE
     ACCESSFIELD    = <3:5>#,
       FCB^ACCESSFIELD          = FCB.OPENACCESS.ACCESSFIELD#;
!
! - DEVINFO.
!
DEFINE
     FILETYPE   =   <0:3>#,
       FCB^FILETYPE             = FCB.DEVINFO.FILETYPE#;
LITERAL
     UNSTR      = 0,
     ESEQ       = 1,
     REL        = 2,
     KSEQ       = 3,
     EDIT       = 4,
     ODDUNSTR   = 8;
DEFINE
     STRUCTFILE  = <2:3>#,       ! <>0 means structured file.
       FCB^STRUCTFILE           = FCB^DEVINFO.STRUCTFILE#;

DEFINE
     DEVTYPE    =   <4:9>#,
       FCB^DEVTYPE              = FCB.DEVINFO.DEVTYPE#;
LITERAL
     PROCESS    = 0,
     OPERATOR   = 1,
     RECEIVE    = 2,
     DISC       = 3,
     MAGTAPE    = 4,
     PRINTER    = 5,
     TERMINAL   = 6,
     DATACOMM   = 7,
     CARDRDR    = 8;
DEFINE
     DEVSUBTYPE = <10:15>#,
       FCB^DEVSUBTYPE           = FCB.DEVINFO.DEVSUBTYPE#;
```

```
!
! OPEN FLAGS. ( FCB.OPENFLAGS1 )
!
DEFINE
     FILECREATED       =  <0>#,  ! new file created.
       FCB^FILECREATED          = FCB.OPENFLAGS1.FILECREATED#;
DEFINE
     ACCESS         =    <1:3>#,  ! access mode.
       FCB^ACCESS               = FCB.OPENFLAGS1.ACCESS#;
LITERAL
       READACCESS       = 1,
       WRITEACCESS      = 2,
       READWRITEACCESS  = 3;
!
! allowable open flags 1 settings.
                             !          111111
LITERAL                      !0123456789012345
     ALLOWED^OPENFLAGS1 = %B0000111111111111;
!
! default open flags 1 settings.
                             !          111111
LITERAL                      !0123456789012345
     DEFAULT^OPENFLAGS1 = %B0000000000000000;
!
! OPEN FLAGS. ( FCB.OPENFLAGS2 )
!
DEFINE
     ABORTONOPENERROR = <15>#,  ! abend on fatal error during open.
       FCB^ABORTONOPENERROR    = FCB.OPENFLAGS2.ABORTONOPENERROR#;
DEFINE
     ABORTONXFERERROR = <14>#,  ! abend on fatal error during data
                                ! transfer.
       FCB^ABORTONXFERERROR    = FCB.OPENFLAGS2.ABORTONXFERERROR#;
DEFINE
     PRINTERRMSG       = <13>#,  ! print error message on fatal error.
       FCB^PRINTERRMSG          = FCB.OPENFLAGS2.PRINTERRMSG#;
DEFINE
     AUTOCREATE        = <12>#,  ! create a file if write access.
       ! 0 = don't.
       ! 1 = do.
       FCB^AUTOCREATE           = FCB.OPENFLAGS2.AUTOCREATE#;
DEFINE
     FILEMUSTBENEW     = <11>#,  ! if autocreate = 1, no such file may
                                 ! currently exist.
       ! 0 = old file is allowed.
       ! 1 = file must be new.
       FCB^FILEMUSTBENEW        = FCB.OPENFLAGS2.FILEMUSTBENEW#;
DEFINE
     WRITEPURGEDATA    = <10>#,  ! purge existing data.
       ! 0 = APPEND.
       ! 1 = PURGEDATA.
       FCB^WRITEPURGEDATA       = FCB.OPENFLAGS2.WRITEPURGEDATA#;
DEFINE
     AUTOTOF           =  <9>#,  ! auto page eject on open for
```

```
                                  ! printer/process.
       ! 0 = NO
       ! 1 = YES
       FCB^AUTOTOF                = FCB.OPENFLAGS2.AUTOTOF#;
DEFINE
     NOWAITIO          =  <8>#,   ! open with no-wait depth of 1.
       ! 0 = WAIT.
       ! 1 = NO-WAIT.
       FCB^NOWAITIO               = FCB.OPENFLAGS2.NOWAITIO#;
DEFINE
     BLOCKEDIO         =  <7>#,   ! blocked i/o.
       ! 0 = NOT BLOCKED
       ! 1 = BLOCKED
       FCB^BLOCKEDIO              = FCB.OPENFLAGS2.BLOCKEDIO#;
DEFINE
     VARFORMAT         =  <6>#,   ! variable length records.
       ! 0 = FIXED LENGTH
       ! 1 = VARIABLE LENGTH
       FCB^VARFORMAT              = FCB.OPENFLAGS2.VARFORMAT#;
DEFINE
     READTRIM          =  <5>#,   ! trim trailing blanks.
       ! 0 = NOTRIM
       ! 1 = TRIM
       FCB^READTRIM               = FCB.OPENFLAGS2.READTRIM#;
DEFINE
     WRITETRIM         =  <4>#,   ! trim trailing blanks.
       ! 0 = NOTRIM
       ! 1 = TRIM
       FCB^WRITETRIM              = FCB.OPENFLAGS2.WRITETRIM#;
DEFINE
     WRITEFOLD         =  <3>#,   ! fold write transfers greater than
       ! 0 = TRUNCATE.               write record length bytes into
       ! 1 = FOLD.                   multiple records.
       FCB^WRITEFOLD              = FCB.OPENFLAGS2.WRITEFOLD#;
DEFINE
     WRITEPAD          =  <2>#,   ! pad record with trailing blanks.
       FCB^WRITEPAD               = FCB.OPENFLAGS2.WRITEPAD#;
DEFINE
     CRLFBREAK         =  <1>#,   ! carriage return/line feed on break.
       ! 0 = NO CRLF ON BREAK.
       ! 1 = CRLF ON BREAK.
       FCB^CRLFBREAK              = FCB.OPENFLAGS2.CRLFBREAK#;
!
! allowable open flags 2 settings.
                           !          111111
LITERAL                    !0123456789012345
     ALLOWED^OPENFLAGS2 = %B1111111111111111;
!
! default open flags 2 settings.
                           !          111111
LITERAL                    !0123456789012345
     DEFAULT^OPENFLAGS2 = %B0101110001001111;
!
! TRANSFER CONTROL ( FCB.XFERCNTL1 )
```

```
!
DEFINE
     ERRORSET         = <0>#,
       ! ERROR SET INTO FCB VIA SET^FILE.
       FCB^ERRORSET             = FCB.XFERCNTL1.ERRORSET#;
DEFINE
     READIOTYPE       = <1:3>#,
       ! 0 = READ
       ! 1 = READUPDATE/REPLY
       ! 2 = EDITREAD
       ! 3 = WRITEREAD
       ! 7 = INVALID
       FCB^READIOTYPE           = FCB.XFERCNTL1.READIOTYPE#;
LITERAL
       STANDARDTYPE    = 0,
       RECEIVETYPE     = 1,
       EDITTYPE        = 2,
       INTERACTIVETYPE = 3,
       INVALIDTYPE     = 7;
DEFINE
     WRITEIOTYPE      = <4:6>#,
       ! 0 = WRITE
       ! 1 = READUPDATE/REPLY
       ! 2 = EDITWRITE
       ! 7 = INVALID
       FCB^WRITEIOTYPE          = FCB.XFERCNTL1.WRITEIOTYPE#;
DEFINE
     SYSBUFLEN        = <7:8>#,  ! system buffer length / 1024.
       FCB^SYSBUFLEN            = FCB.XFERCNTL1.SYSBUFLEN#;
DEFINE
     PROMPT          = <9:15>#,  ! interactive prompt character.
       FCB^PROMPT               = FCB.XFERCNTL1.PROMPT#;
!
! TRANSFER CONTROL ( FCB.XFERCNTL2 )
!
DEFINE
     PHYSIOOUT          = <0>#,  ! physical (read/write) i/o
                                 ! outstanding.
       FCB^PHYSIOOUT            = FCB.XFERCNTL2.PHYSIOOUT#,
     READIOOUT          = <1>#,  ! logical read i/o outstanding.
       FCB^READIOOUT            = FCB.XFERCNTL2.READIOOUT#,
     WRITEIOOUT         = <2>#,  ! logical write i/o outstanding.
       FCB^WRITEIOOUT           = FCB.XFERCNTL2.WRITEIOOUT#,
     LOGIOOUT         = <1:2>#,  ! logical i/o outstanding.
       FCB^LOGIOOUT             = FCB.XFERCNTL2.LOGIOOUT#,
     WRITEFLUSH        =  <3>#,  ! block buffer flush operation in
                                 ! progress.
       FCB^WRITEFLUSH           = FCB.XFERCNTL2.WRITEFLUSH#,
     RETRYCOUNT       = <4:5>#,  ! i/o retry counter.
       FCB^RETRYCOUNT           = FCB.XFERCNTL2.RETRYCOUNT#,
     NOPARTIALREC       = <6>#,  ! blocks contain only full records.
       FCB^NOPARTIALREC         = FCB.XFERCNTL2.NOPARTIALREC#;
!
! TRANSFER CONTROL ( FCB.XFERCNTL2 )
```

```
!
! -- EDIT READ/WRITE CONTROL
!
DEFINE
     EDDIRWIP        =     <7>#, ! directory write in progress.
       FCB^EDDIRWIP            = FCB.XFERCNTL2.EDDIRWIP#,
     EDHALFSECTCNT   =  <8:11>#, ! number of half sectors written in
                                 ! current data page after next
                                 ! physical write.
       FCB^EDHALFSECTCNT       = FCB.XFERCNTL2.EDHALFSECTCNT#,
     EDDATABUFLEN    = <12:15>#, ! edit data buf size '>>'
                                 ! EDDBUFSHIFT (8).
       FCB^EDDATABUFLEN        = FCB.XFERCNTL2.EDDATABUFLEN#,
     EDREPOSITION    =     <7>#, ! user is repositioning edit file
                                 ! (read op).
       FCB^EDREPOSITION        = FCB.XFERCNTL2.EDREPOSITION#;
!
! WRITE^FILE CONTROL OPERATION IN PROGRESS ( FCB.PHYSXFERCNT )
!
DEFINE
     CNTLINPROGRESS =     <0>#,
       FCB^CNTLINPROGRESS      = FCB.PHYSXFERCNT.CNTLINPROGRESS#,
     FORMSCNTLOP     =  <1:15>#,
       FCB^FORMSCNTLOP         = FCB.PHYSXFERCNT.FORMSCNTLOP#;
!
! BREAK CONTROL ( COMMFCB.BRKCNTL )
!
DEFINE
     BRKLASTMODE     = <0>#,     ! last break mode from SETMODE.
       COMMFCB^BRKLASTMODE     = COMMFCB.BRKCNTL.BRKLASTMODE#,
     BRKHIT          = <1>#,     ! BREAK key has been typed but not
                                 ! tested.
       COMMFCB^BRKHIT          = COMMFCB.BRKCNTL.BRKHIT#,
     BRKFLUSH        = <2>#,     ! flush $RECEIVE BREAK message.
       COMMFCB^BRKFLUSH        = COMMFCB.BRKCNTL.BRKFLUSH#,
     BRKSTOLEN       = <3>#,     ! BREAK stolen away by another
                                 ! process.
       COMMFCB^BRKSTOLEN       = COMMFCB.BRKCNTL.BRKSTOLEN#,
     BRKLDN          = <8:15>#, ! logical device number of terminal.
       COMMFCB^BRKLDN          = COMMFCB.BRKCNTL.BRKLDN#,
     COMMFCB^BRKARMED =          ! BREAK is armed.
       COMMFCB^BRKFCBADDR#;
!
! $RECEIVE CONTROL ( COMMFCB.RCVCNTL )
!
DEFINE
     RCVDATAOPEN      =    <0>#, ! $RECEIVE has been opened for data
                                 ! transfer.
       COMMFCB^RCVDATAOPEN     = COMMFCB.RCVCNTL.RCVDATAOPEN#,
     RCVBRKOPEN       =    <1>#, ! $RECEIVE has been opened for BREAK
                                 ! message reception.
       COMMFCB^RCVBRKOPEN      = COMMFCB.RCVCNTL.RCVBRKOPEN#,
     RCVOPENCNT       = <2:3>#, ! count of OPEN messages received.
       COMMFCB^RCVOPENCNT      = COMMFCB.RCVCNTL.RCVOPENCNT#,
```

```
      RCVSTATE          =    <4>#,
        ! 0 = NEED READUPDATE.
        ! 1 = NEED REPLY.
        COMMFCB^RCVSTATE        = COMMFCB.RCVCNTL.RCVSTATE#,
      RCVUSEROPENREPLY =    <5>#, ! user will reply to OPEN messages.
        ! 0 = SIO REPLIES.
        ! 1 = USER REPLIES.
        COMMFCB^RCVUSEROPENREPLY = COMMFCB.RCVCNTL.RCVUSEROPENREPLY#,
      RCVPSUEDOEOF      =    <6>#, ! pseudo-EOF. (N/A if user wants
                                   ! CLOSE messages)
        ! 0 = EAT CLOSE MESSAGE.
        ! 1 = TURN LAST CLOSE MESSAGE INTO EOF.
        COMMFCB^RCVPSUEDOEOF    = COMMFCB.RCVCNTL.RCVPSUEDOEOF#,
      MONCPUMSG         = <2:3>#, ! user CPU Up/Down messages.
        COMMFCB^MONCPUMSG       = COMMFCB.SYSMSGS1.MONCPUMSG#,
      OPENMSG           =   <14>#, ! user wants OPEN messages.
        COMMFCB^OPENMSG         = COMMFCB.SYSMSGS2.OPENMSG#,
      CLOSEMSG          =   <15>#, ! user wants CLOSE messages.
        COMMFCB^CLOSEMSG        = COMMFCB.SYSMSGS2.CLOSEMSG#;
!
! COMMON CONTROL ( COMMFCB.COMMCNTL )
!
DEFINE
      CREATEINPROGRESS  = <0>#,  ! 1 during call to OPEN^FILE while
                                 ! creating.
        COMMFCB^CREATEINPROGRESS = COMMFCB.COMMCNTL.CREATEINPROGRESS#,
      OPENINPROGRESS    = <1>#,  ! 1 during call to OPEN^FILE.
        COMMFCB^OPENINPROGRESS = COMMFCB.COMMCNTL.OPENINPROGRESS#,
      OPTYPE          = <0:1>#,  ! operation type.
        COMMFCB^OPTYPE          = COMMFCB.COMMCNTL.OPTYPE#,
      DEFAULTERRFILE    = <2>#,  ! defines default error reporting
                                 ! file.
        ! 0 = home terminal.
        ! 1 = operator ($0).
        COMMFCB^DEFAULTERRFILE = COMMFCB.COMMCNTL.DEFAULTERRFILE#,
      TRACEBACK         = <3>#,  ! 1 = trace back to caller's P when
                                 ! printing an error message.
        COMMFCB^TRACEBACK       = COMMFCB.COMMCNTL.TRACEBACK#;
```

APPENDIX F

ASCII CHARACTER SET

The table on the following pages lists the characters in the ASCII character set, with their meanings and their octal values.

Two octal characters fit into a 16-bit word. Because of shifting, the octal value of a character depends on whether it is in the left-hand or the right-hand byte of the word. To determine the octal value of a word containing two ASCII characters, add the "left byte" value of the left-hand character to the "right byte" value of the right-hand character.

For example, the octal value of "AB" is:

```
  040400   ("left byte" value of "A")
+ 000102   ("right byte" value of "B")
---------
  040502   (total)
```

Character	Octal Value (left byte)	Octal Value (right byte)	Meaning
NUL	000000	000000	Null
SOH	000400	000001	Start of heading
STX	001000	000002	Start of text
ETX	001400	000003	End of text
EOT	002000	000004	End of transmission
ENQ	002400	000005	Enquiry
ACK	003000	000006	Acknowledge
BEL	003400	000007	Bell
BS	004000	000010	Backspace
HT	004400	000011	Horizontal tabulation
LF	005000	000012	Line feed
VT	005400	000013	Vertical tabulation
FF	006000	000014	Form feed
CR	006400	000015	Carriage return
SO	007000	000016	Shift out
SI	007400	000017	Shift in
DLE	010000	000020	Data link escape
DC1	010400	000021	Device control 1
DC2	011000	000022	Device control 2
DC3	011400	000023	Device control 3
DC4	012000	000024	Device control 4
NAK	012400	000025	Negative acknowledge
SYN	013000	000026	Synchronous idle
ETB	013400	000027	End of transmission block
CAN	014000	000030	Cancel
EM	014400	000031	End of medium
SUB	015000	000032	Substitute
ESC	015400	000033	Escape
FS	016000	000034	File separator
GS	016400	000035	Group separator
RS	017000	000036	Record separator
US	017400	000037	Unit separator
SP	020000	000040	Space
!	020400	000041	Exclamation point
"	021000	000042	Quotation mark
#	021400	000043	Number sign
$	022000	000044	Dollar sign
%	022400	000045	Percent sign
&	023000	000046	Ampersand
'	023400	000047	Apostrophe

Character	Octal Value (left byte)	Octal Value (right byte)	Meaning
(	024000	000050	Opening parenthesis
)	024400	000051	Closing parenthesis
*	025000	000052	Asterisk
+	025400	000053	Plus
,	026000	000054	Comma
-	026400	000055	Hyphen (minus)
.	027000	000056	Period (decimal point)
/	027400	000057	Right slant
0	030000	000060	Zero
1	030400	000061	One
2	031000	000062	Two
3	031400	000063	Three
4	032000	000064	Four
5	032400	000065	Five
6	033000	000066	Six
7	033400	000067	Seven
8	034000	000070	Eight
9	034400	000071	Nine
:	035000	000072	Colon
;	035400	000073	Semicolon
<	036000	000074	Less than
=	036400	000075	Equals
>	037000	000076	Greater than
?	037400	000077	Question mark
@	040000	000100	Commercial at
A	040400	000101	Upper-case A
B	041000	000102	Upper-case B
C	041400	000103	Upper-case C
D	042000	000104	Upper-case D
E	042400	000105	Upper-case E
F	043000	000106	Upper-case F
G	043400	000107	Upper-case G
H	044000	000110	Upper-case H
I	044400	000111	Upper-case I
J	045000	000112	Upper-case J
K	045400	000113	Upper-case K
L	046000	000114	Upper-case L
M	046400	000115	Upper-case M
N	047000	000116	Upper-case N
O	047400	000117	Upper-case O

Character	Octal Value (left byte)	Octal Value (right byte)	Meaning
P	050000	000120	Upper-case P
Q	050400	000121	Upper-case Q
R	051000	000122	Upper-case R
S	051400	000123	Upper-case S
T	052000	000124	Upper-case T
U	052400	000125	Upper-case U
V	053000	000126	Upper-case V
W	053400	000127	Upper-case W
X	054000	000130	Upper-case X
Y	054400	000131	Upper-case Y
Z	055000	000132	Upper-case Z
[	055400	000133	Opening bracket
\	056000	000134	Left slant
]	056400	000135	Closing bracket
^	057000	000136	Circumflex
_	057400	000137	Underscore
`	060000	000140	Grave accent
a	060400	000141	Lower-case a
b	061000	000142	Lower-case b
c	061400	000143	Lower-case c
d	062000	000144	Lower-case d
e	062400	000145	Lower-case e
f	063000	000146	Lower-case f
g	063400	000147	Lower-case g
h	064000	000150	Lower-case h
i	064400	000151	Lower-case i
j	065000	000152	Lower-case j
k	065400	000153	Lower-case k
l	066000	000154	Lower-case l
m	066400	000155	Lower-case m
n	067000	000156	Lower-case n
o	067400	000157	Lower-case o
p	070000	000160	Lower-case p
q	070400	000161	Lower-case q
r	071000	000162	Lower-case r
s	071400	000163	Lower-case s
t	072000	000164	Lower-case t
u	072400	000165	Lower-case u
v	073000	000166	Lower-case v
w	073400	000167	Lower-case w

Character	Octal Value (left byte)	Octal Value (right byte)	Meaning
x	074000	000170	Lower-case x
y	074400	000171	Lower-case y
z	075000	000172	Lower-case z
{	075400	000173	Opening brace
|	076000	000174	Vertical line
}	076400	000175	Closing brace
~	077000	000176	Tilde
DEL	077400	000177	Delete

INDEX

READER'S COMMENTS

Tandem welcomes your feedback on the quality and usefulness of its publications. Please indicate a specific *section* and *page* number when commenting on any manual. Does this manual have the desired completeness and flow of organization? Are the examples clear and useful? Is it easily understood? Does it have obvious errors? Are helpful additions needed?

Title of manual(s): _____

FOLD ➤

FOLD ➤

FROM:

Name _____

Company _____

Address _____

City/State _____ Zip _____

A written response is requested, yes no ?

FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 482 CUPERTINO, CA. U.S.A.

POSTAGE WILL BE PAID BY ADDRESSEE

TANDEM
COMPUTERS

19333 Vallco Parkway
Cupertino, CA U.S.A. 95014
Attn: Technical Communications—Software

FOLD

STAPLE HERE