GUARDIAN
Operating System
Programming
Manual
Volume 2
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

Copyright (c) 1982 by Tandem Computers Incorporated.

All rights reserved. No part of this document may be reproduced in any form, including photocopying or translation to another programming language, without the prior written consent of Tandem Computers Incorporated.

The following are trademarks of Tandem Computers Incorporated: Tandem, NonStop, AXCESS, DYNABUS, ENABLE, ENCOMPASS, ENFORM, ENSCRIBE, ENVISION, ENVOY, EXCHANGE, EXPAND, GUARDIAN, PATHWAY, TGAL, XRAY.
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
- 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

#### SECTION 1. INTRODUCTION TO THE GUARDIAN OPERATING SYSTEM

- Process Control ............................................. 1-1
- Process Structure .......................................... 1-5
- Process Pairs ............................................... 1-7
- Process Control Functions ................................ 1-8
- File System ................................................. 1-9
- Utility Procedures .......................................... 1-10
- System Messages ............................................ 1-13
- Checkpointing Facility (Fault-Tolerant Programming) 1-14
- Traps and Trap Handling .................................... 1-16
- Security ...................................................... 1-17
- Command Interpreter ........................................ 1-18
- Debug Facility ............................................... 1-18
- External Declarations for Operating System Procedures 1-18

#### SECTION 2. FILE SYSTEM

- INTRODUCTION ............................................. 2.1-1
- Files .......................................................... 2.1-1
- Disc Files ................................................... 2.1-1
- Non-Disc Devices .......................................... 2.1-3
- Processes (Interprocess Communication) ................. 2.1-4
- Operator Console .......................................... 2.1-7
- File Access .................................................. 2.1-7
- Disc Files ................................................... 2.1-8
- Terminals .................................................... 2.1-10
- Processes .................................................... 2.1-10
- Access Coordination Among Multiple Accessors ............ 2.1-11
- Locking ....................................................... 2.1-12
- Wait/No-Wait I/O ............................................ 2.1-13
- File System Implementation ................................ 2.1-16
- File and I/O System Structure ............................. 2.1-16
- File System Procedure Execution .......................... 2.1-21
- File Open ..................................................... 2.1-21
- File Transfers ............................................... 2.1-24
- Buffering ...................................................... 2.1-26
- File Close ..................................................... 2.1-27
- Automatic Path Error Recovery for Disc Files .......... 2.1-28
- Mirror Volumes .............................................. 2.1-34
Error Indication .............................................. 2.1-35
Error Recovery .............................................. 2.1-37

FILE NAMES ..................................................... 2.2-1
 Disc File Names .............................................. 2.2-2
  Volume Name .............................................. 2.2-2
  Subvol Name .............................................. 2.2-2
  Disc File Name ........................................... 2.2-2
  Temporary File Name ...................................... 2.2-2
  Examples .................................................. 2.2-3
 Device Names ................................................ 2.2-3
 Logical Device Numbers .................................... 2.2-3
 $RECEIVE ....................................................... 2.2-3
 Process ID .................................................... 2.2-4
  Timestamp Form ............................................ 2.2-4
  Process Name Form ........................................ 2.2-4
 Obtaining a Process ID .................................... 2.2-5
 Examples ..................................................... 2.2-5
 $0 .............................................................. 2.2-6
 Network File Names .......................................... 2.2-7
  Process ID, Network Form ................................ 2.2-8

FILE SYSTEM PROCEDURES ...................................... 2.3-1
 Characteristics .............................................. 2.3-3
  For Procedure Usage by Device Type ........................ 2.3-3
 Completion ................................................. 2.3-4
 <file number> Parameters .................................. 2.3-4
 <tag> Parameters ........................................... 2.3-5
 <buffer> Parameter ........................................ 2.3-5
 <transfer count> Parameter ................................ 2.3-5
 Condition Codes ............................................ 2.3-6
 Errors ....................................................... 2.3-6
 Access Mode and Security Checking for Disc Files ......... 2.3-6
 AWAITIO Procedure (all files) ............................... 2.3-7
 CANCEL Procedure (all files) ............................... 2.3-11
 CANCELREQ Procedure (all files) ............................ 2.3-12
 CLOSE Procedure (all files) ................................ 2.3-13
 CONTROL Procedure (all files) .............................. 2.3-15
 CONTROLBUF Procedure (all files) ......................... 2.3-20
 CREATE Procedure (disc files) .............................. 2.3-23
 DEVICEINFO Procedure (all files) .......................... 2.3-26
 EDITREAD Procedure (edit-type files) ....................... 2.3-30
 EDITREADINIT Procedure (edit-type files) .................. 2.3-34
 FILEERROR Procedure (all files) ........................... 2.3-36
 FILEINFO Procedure (all files) ............................ 2.3-39
 FNAMECOLLAPSE Procedure (all files) ....................... 2.3-43
 FNAMECOMPARE Procedure (all files) ......................... 2.3-45
 FNAMEXPAND Procedure (all files) ........................... 2.3-48
 GETDEVNAME Procedure (disc files and non-disc devices) .. 2.3-52
 GETSYSTEMNAME Procedure .................................. 2.3-54
 LASTRECEIVE Procedure ($RECEIVE file) .................... 2.3-55
 LOCATESYSTEM Procedure .................................... 2.3-57
 LOCKFILE Procedure (disc files) ........................... 2.3-58
 MONITORNET Procedure ..................................... 2.3-61
MONITORNEW Procedure (NonStop II systems only).......................2.3-62
NEXTFILENAME Procedure (disc files)......................................2.3-63
OPEN Procedure (all files)....................................................2.3-65
POSITION Procedure (disc files).............................................2.3-73
PURGE Procedure (disc files)...................................................2.3-75
READ Procedure (all files).....................................................2.3-76
READUPDATE Procedure (disc files and $RECEIVE files)..............2.3-79
RECEIVEINFO Procedure ($RECEIVE file)....................................2.3-82
REFRESH Procedure (disc files)...............................................2.3-85
REMOTEPROCESSORSTATUS Procedure.........................................2.3-86
RENAME Procedure (disc files)................................................2.3-88
REPOSITION Procedure ($RECEIVE file).......................................2.3-89
WRITE Procedure (all files)....................................................2.3-108
WRITEUPDATE Procedure (disc and magnetic tape files)..............2.3-112

FILE SYSTEM ERRORS AND ERROR RECOVERY............................................2.4-1
Error List.......................................................................................2.4-2
Error Recovery..............................................................................2.4-29
  Device.........................................................................................2.4-29
  Path Errors (Errors 200-255)...................................................2.4-29
  No-Wait I/O.................................................................................2.4-32
File System Error Messages on the Operator Console..................2.4-32

TERMINALS: CONVERSATIONAL MODE/PAGE MODE.................................2.5-1
General Characteristics of Terminals..............................................2.5-1
Summary of Applicable Procedures...............................................2.5-3
Accessing Terminals........................................................................2.5-4
  Transfer Termination when Reading............................................2.5-5
Transfer Modes...............................................................................2.5-6
  Conversational Mode......................................................................2.5-8
  Page Mode.....................................................................................2.5-16
Transparency Mode (Interrupt Character Checking Disabled)........2.5-22
Checksum Processing (Read Termination on ETX Character)............2.5-22
Echo...............................................................................................2.5-22
Timeouts.........................................................................................2.5-23
Modems............................................................................................2.5-23
Break Feature...............................................................................2.5-25
  BREAK System Message.............................................................2.5-26
  Using BREAK (Single Process per Terminal)..............................2.5-26
  Using BREAK (More than One Process per Terminal)...................2.5-28
  Break Mode...................................................................................2.5-29
Error Recovery...............................................................................2.5-34
  Operation Timed Out (Error 40)...................................................2.5-34
  BREA(Errors 110 and 111)............................................................2.5-34
  Preempted by Operator Message (Error 112)................................2.5-35
  Modem Error (Error 140).............................................................2.5-36
INITIALIZER Procedure .................................................. 4-13
LASTADDR Procedure .................................................... 4-17
NUMIN Procedure .......................................................... 4-18
NUMOUT Procedure ...................................................... 4-21
SHIFTSTRING Procedure ................................................ 4-23
TIME Procedure ........................................................... 4-24
TIMESTAMP Procedure .................................................. 4-25
TOSVERSION Procedure ................................................. 4-26

SECTION 5. CHECKPOINTING FACILITY ............................... 5.1-1
INTRODUCTION ............................................................. 5.1-1
  Overview of Checkpointing Procedures .......................... 5.1-1
  Overview of NonStop Programs .................................... 5.1-2
  Overview of Checkpointing ........................................ 5.1-4
    Data Stack ......................................................... 5.1-5
    Data Buffers ..................................................... 5.1-5
    Sync Blocks ...................................................... 5.1-5

CHECKPOINTING PROCEDURES ......................................... 5.2-1
  CHECKCLOSE Procedure ............................................ 5.2-3
  CHECKMONITOR Procedure ....................................... 5.2-5
  CHECKOPEN Procedure ............................................. 5.2-9
  CHECKPOINT Procedure ............................................ 5.2-12
  CHECKPOINTMANY Procedure .................................... 5.2-14
  CHECKSWITCH Procedure ......................................... 5.2-17
  GETSYNCINFO Procedure (disc files) ............................ 5.2-18
  MONITORCPUS Procedure ......................................... 5.2-19
  PROCESSORSTATUS Procedure ................................. 5.2-21
  RESETSYNC Procedure (disc files) ............................ 5.2-22
  SETSYNCINFO Procedure (disc files) ......................... 5.2-23

USING THE CHECKPOINTING FACILITY ................................. 5.3-1
  NonStop Program Structure ...................................... 5.3-1
    Process Startup for Named Process Pairs .................. 5.3-1
    Process Startup for Non-Named Process Pairs ............. 5.3-9
    Main Processing Loop ......................................... 5.3-13
  File Open .......................................................... 5.3-13
  Checkpointing ..................................................... 5.3-14
    Guidelines for Checkpointing ................................ 5.3-15
    Example of Where Checkpoints Should Occur .............. 5.3-17
    Checkpointing Multiple Disc Updates ..................... 5.3-21
    Considerations for No-Wait I/O ................................ 5.3-21
    Action for CHECKPOINT Failure ............................ 5.3-21
  System Messages ................................................ 5.3-22
    Recommended Action ........................................... 5.3-23
    Takeover by Backup .......................................... 5.3-25
    Opening a File During Processing .......................... 5.3-27
    Creation of a Descendant Process (Pair) ................ 5.3-28

ADVANCED CHECKPOINTING ........................................... 5.4-1
  Backup Open ...................................................... 5.4-1
  File Synchronization Information ............................ 5.4-2
## SECTION 6. TRAPS AND TRAP HANDLING

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Traps</td>
<td>6-1</td>
</tr>
<tr>
<td>Trap Handling</td>
<td>6-1</td>
</tr>
<tr>
<td>ARMTRAP Procedure</td>
<td>6-4</td>
</tr>
<tr>
<td>Example</td>
<td>6-7</td>
</tr>
</tbody>
</table>

## SECTION 7. SECURITY SYSTEM

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTRODUCTION</td>
<td>7.1-1</td>
</tr>
<tr>
<td>System Users</td>
<td>7.1-2</td>
</tr>
<tr>
<td>Defining Users</td>
<td>7.1-3</td>
</tr>
<tr>
<td>Naming Conventions</td>
<td>7.1-4</td>
</tr>
<tr>
<td>Logging On</td>
<td>7.1-4</td>
</tr>
<tr>
<td>Passwords</td>
<td>7.1-5</td>
</tr>
<tr>
<td>Accessor ID</td>
<td>7.1-5</td>
</tr>
<tr>
<td>Disc File Security</td>
<td>7.1-6</td>
</tr>
<tr>
<td>Adopting a Program File's Owner ID</td>
<td>7.1-8</td>
</tr>
<tr>
<td>Licensing</td>
<td>7.1-9</td>
</tr>
<tr>
<td>Interface to the Security System</td>
<td>7.1-10</td>
</tr>
<tr>
<td>Command Interpreter Interface</td>
<td>7.1-10</td>
</tr>
<tr>
<td>FUP Interface</td>
<td>7.1-10</td>
</tr>
<tr>
<td>Programmatic Interface</td>
<td>7.1-11</td>
</tr>
<tr>
<td>Operational Limitations</td>
<td>7.1-11</td>
</tr>
<tr>
<td>Network Security</td>
<td>7.1-14</td>
</tr>
<tr>
<td>Global Knowledge of User ID's</td>
<td>7.1-14</td>
</tr>
<tr>
<td>Remote Passwords</td>
<td>7.1-15</td>
</tr>
<tr>
<td>Process Access</td>
<td>7.1-17</td>
</tr>
<tr>
<td>Programmatically Logging ID's</td>
<td>7.1-17</td>
</tr>
</tbody>
</table>

### SECURITY SYSTEM PROGRAMMATIC INTERFACE

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATORACCESSID Procedure</td>
<td>7.2-2</td>
</tr>
<tr>
<td>PROCESSACCESSID Procedure</td>
<td>7.2-3</td>
</tr>
<tr>
<td>Functions for SETMODE and SETMODENOWAIT Procedures</td>
<td>7.2-4</td>
</tr>
<tr>
<td>SETSTOP Procedure</td>
<td>7.2-7</td>
</tr>
<tr>
<td>USERIDTOUSERNAME Procedure</td>
<td>7.2-8</td>
</tr>
<tr>
<td>USERNAMETOUSERID Procedure</td>
<td>7.2-9</td>
</tr>
<tr>
<td>VERIFYUSER Procedure</td>
<td>7.2-10</td>
</tr>
</tbody>
</table>

## SECTION 8. MEMORY MANAGEMENT PROCEDURES

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>MANAGING EXTENDED SEGMENTS (NonStop II systems only)</td>
<td>8.1-1</td>
</tr>
<tr>
<td>Segmented Memory</td>
<td>8.1-1</td>
</tr>
<tr>
<td>Space Management Within a Segment</td>
<td>8.1-3</td>
</tr>
<tr>
<td>ALLOCATESEGMENT Procedure</td>
<td>8.1-4</td>
</tr>
<tr>
<td>DEALLOCATESEGMENT Procedure</td>
<td>8.1-6</td>
</tr>
<tr>
<td>DEFINEPOOL Procedure</td>
<td>8.1-7</td>
</tr>
<tr>
<td>GETPOOL Procedure</td>
<td>8.1-8</td>
</tr>
<tr>
<td>PUTPOOL Procedure</td>
<td>8.1-9</td>
</tr>
<tr>
<td>USESEGMENT Procedure</td>
<td>8.1-10</td>
</tr>
</tbody>
</table>

### ADVANCED MEMORY MANAGEMENT

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOCKDATA Procedure (NonStop systems only)</td>
<td>8.2-2</td>
</tr>
<tr>
<td>LOCKMEMORY Procedure (NonStop II systems only)</td>
<td>8.2-5</td>
</tr>
<tr>
<td>UNLOCKMEMORY Procedure (NonStop II systems only)</td>
<td>8.2-8</td>
</tr>
</tbody>
</table>

xiv
SECTION 9. SEQUENTIAL I/O PROCEDURES ................................................. 9-1
  CHECK~BREAK Procedure ........................................................................ 9-4
  CHECK~FILE Procedure ............................................................................ 9-5
  CLOSE~FILE Procedure ............................................................................ 9-12
  GIVE~BREAK Procedure ........................................................................... 9-14
  OPEN~FILE Procedure .............................................................................. 9-15
  READ~FILE Procedure ............................................................................. 9-21
  SET~FILE Procedure ................................................................................ 9-23
  TAKE~BREAK Procedure .......................................................................... 9-33
  WAIT~FILE Procedure ............................................................................... 9-34
  WRITE~FILE Procedure ............................................................................ 9-36

Errors ........................................................................................................ 9-38

FCB Structure .............................................................................................. 9-41
  Initializing the File FCB .......................................................................... 9-42

Interface With INITIALIZER and ASSIGN Messages ..... 9-46
  INITIALIZER-Related Defines ................................................................. 9-46
  Usage Example ......................................................................................... 9-50

Usage Example Without INITIALIZER Procedure .................................... 9-54

NO~ERROR Procedure .............................................................................. 9-56

$RECEIVE Handling .................................................................................. 9-60
  $RECEIVE Data Transfer Protocol .......................................................... 9-60

No-Wait I/O ................................................................................................. 9-63

Summary of FCB Attributes ....................................................................... 9-64

SECTION 10. FORMATTER ............................................................................. 10-1

FORMATCONVERT Procedure .................................................................. 10-2

FORMATDATA Procedure .......................................................................... 10-5

Errors ........................................................................................................... 10-9

Example ...................................................................................................... 10-10

Format-Directed Formatting ....................................................................... 10-13
  Format Characteristics ............................................................................ 10-14
  Edit Descriptors ...................................................................................... 10-17
  Non-Repeatable Edit Descriptors ............................................................ 10-20
    Tabulation Descriptors ......................................................................... 10-20
    Literal Descriptors .............................................................................. 10-21
    Scale Factor Descriptor (P) ................................................................. 10-22
    Optional Plus Descriptors (S, SP, SS) ................................................ 10-23
    Blank Descriptors (BN, BZ) ................................................................. 10-24
    Buffer Control Descriptors (/, ::) ......................................................... 10-24

Repeatable Edit Descriptors ...................................................................... 10-26
  "A" Edit Descriptor .................................................................................. 10-26
  "D" Edit Descriptor .................................................................................. 10-28
  "E" Edit Descriptor .................................................................................. 10-28
  "F" Edit Descriptor .................................................................................. 10-31
  "G" Edit Descriptor .................................................................................. 10-32
  "I" Edit Descriptor .................................................................................. 10-34
  "L" Edit Descriptor .................................................................................. 10-35
  "M" Edit Descriptor .................................................................................. 10-37

Modifiers ..................................................................................................... 10-40
  Field Blanking Modifiers (BN, BZ) ........................................................ 10-40
  Fill Character Modifier (FL) .................................................................. 10-40
  Overflow Character Modifier (OC) ........................................................ 10-41
  Justification Modifiers (LJ, RJ) .............................................................. 10-41
  Symbol Substitution Modifier (SS) ......................................................... 10-42
FIGURES

Volume 1

1-1. GUARDIAN Operating System: Mirror Volumes......................1-2
1-2. A Primary/Backup Process Pair........................................1-9
1-3. Files.................................................................................1-11
1-4. Checkpointing.................................................................1-15
1-5. Files Open by a Primary/Backup Process Pair.........................1-16
2-1. Disc File Organization.....................................................2.1-2
2-3. Communication with a Process Pair via Process Name.............2.1-6
2-4. $RECEIVE File....................................................................2.1-6
2-5. Wait versus No-Wait I/O....................................................2.1-13
2-6. No-Wait I/O (Multiple Concurrent Operations).....................2.1-15
2-7. Hardware I/O Structure.....................................................2.1-17
2-8. Primary and Alternate Communication Paths.........................2.1-19
2-9. File System Procedure Execution.......................................2.1-20
2-10. File Open..........................................................................2.1-23
2-11. File Transfer......................................................................2.1-25
2-12. Buffering...........................................................................2.1-26
2-13. Mirror Volume....................................................................2.1-34
2-14. Action of AWAITIO............................................................2.3-10
2-15. File Security Checking.....................................................2.3-70
2-16. File System Path Error Recovery.......................................2.4-30
2-17. Transfer Modes for Terminals............................................2.5-7
2-18. Conversational Mode Interrupt Characters..........................2.5-11
2-19. Page Mode Interrupt Characters.........................................2.5-17
2-20. BREAK: Single Process per Terminal..................................2.5-28
2-21. Break Mode.......................................................................2.5-32
2-22. Exclusive Access Using BREAK........................................2.5-34
2-23. Column-Binary Read Mode for Cards................................2.8-3
2-24. Packed-Binary Read Mode for Cards................................2.8-4
2-25. Link Control Blocks.........................................................2.11-1
2-26. Resident Buffering (NonStop systems only)........................2.11-5
3-1. Program versus Process.....................................................3.1-2
3-2. A Process (NonStop systems)..............................................3.1-3
3-3. A Process (NonStop II systems)..........................................3.1-4
3-4. Process Pairs......................................................................3.1-11
3-5. Home Terminal...................................................................3.1-18
3-6. Effect of STEPMOM............................................................3.2-47
3-7. Execution Priority Example................................................3.4-3
Volume 2

4-1. Last Address.................................................4-17
5-1. A NonStop Program.............................................5.1-3
5-2. Checkpoints and Restart Points..........................5.3-14
5-3. Backup Open by Backup Process.........................5.4-1
7-1. Passing of Accessor ID's..................................7.1-6
7-2. Effect of Adopting a Program File's Owner ID......7.1-9
9-1. FCB Linking.................................................9-41
11-1. File Names..................................................11-3
11-2. Disc File Names.............................................11-5
12-1. "Serveobj" Program.........................................12.1-3
12-2. Request Checkpoint........................................12.1-9

TABLES

Volume 1

2-1. CONTROL Operations........................................2.3-17
2-2. CONTROLBUF Operations....................................2.3-22
2-3. Device Types and Subtypes................................2.3-27
2-4. Exclusion/Access Mode Checking........................2.3-71
2-5. SETMODE Functions..........................................2.3-97
2-6. Path Error Recovery for Devices Other than Discs
and Processes....................................................2.4-31
2-7. Terminal CONTROL and SETMODE Operations..........2.5-37
2-8. Line Printer CONTROL and SETMODE Operations.......2.6-17
2-9. Magnetic Tape CONTROL Operations.....................2.7-17
2-10. ASCII Equivalents to BCD Character Set..............2.7-19

Volume 2

5-1. Action of CHECKMONITOR.................................5.2-7
7-1. Allowability of File Access...............................7.1-8
7-2. Operational Restrictions.................................7.1-12
7-3. SETMODE Functions Related to Security................7.2-4
10-1. Modifiers Usable with Edit Descriptors..............10-43
B-1. File System Error Summary...............................B-3
The following is a summary of the characters and symbols used in the syntax notation in this manual.

<table>
<thead>
<tr>
<th>NOTATION</th>
<th>MEANING</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPPER-CASE</td>
<td>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.</td>
</tr>
<tr>
<td>lower-case</td>
<td>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.</td>
</tr>
<tr>
<td>Brackets</td>
<td>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.</td>
</tr>
<tr>
<td>Braces</td>
<td>A vertically-aligned group of items enclosed in braces, { }, represents a list of selections from which exactly one must be chosen.</td>
</tr>
</tbody>
</table>
| 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:

\[
\begin{align*}
\{ & \text{CALL} \\
& \langle \text{procedure name} \rangle ( \langle \text{parameters} \rangle ) \\
& \langle \text{retval} \rangle := \}
\end{align*}
\]

CALL is a TAL CALL statement.

"\langle \text{retval} \rangle :=" indicates that the procedure is a function procedure (i.e., it returns a value of the indicated \langle \text{type} \rangle when referenced in an expression).

\langle \text{procedure name} \rangle 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:

\[
\langle \text{retval} \rangle, \langle \text{type} \rangle
\]

\langle \text{type} \rangle 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 } [: <num elements>],
{value}

<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
UTILITY PROCEDURES
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:

```plaintext
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:

```plaintext
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:

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

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

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

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

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

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

4-2
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).
UTILITY PROCEDURES
FIXSTRING Procedure

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:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>no change was made to &lt;data&gt;.</td>
</tr>
<tr>
<td>1</td>
<td>a replacement, insertion, or deletion was performed on &lt;data&gt; (see &quot;Considerations&quot; below).</td>
</tr>
</tbody>
</table>
UTILITY PROCEDURES

FIXSTRING Procedure

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.
UTILITY PROCEDURES
FIXSTRING Procedure

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 STRING
<template>: N
<data> on return from FIXSTRING: THIS IS A STRING
UTILITY PROCEDURES
FIXSTRING Procedure

replacement terminated by "/""

<data> on entry to FIXSTRING: THID IS A BAD OLD STRING
$template>: 
$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// dddd rNEW // iN
$data> on return from FIXSTRING: THIS IS A NEW STRING
UTILITY PROCEDURES

FIXSTRING Procedure

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,
   save^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;
```

4-8
FIXSTRING Procedure

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
UTILITY PROCEDURES

FIXSTRING Procedure

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;
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> )
```

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:

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

where

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

- `<passthru>` 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.
UTILITY PROCEDURES
INITIALIZER Procedure

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
```
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.
INITIALIZER Procedure

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 INITIALIZER 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 <rncb> 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".

4-16
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:

\[
\langle \text{last address} \rangle := \text{LASTADDR}
\]

where

\[
\langle \text{last address} \rangle, \text{INT},
\]

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

example:

\[
\text{highest address} := \text{LASTADDR};
\]

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

\[
\text{num pages} := \text{LASTADDR}.<0:5> + 1;
\]

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
UTILITY PROCEDURES
NUMIN Procedure

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> , <base> , <status> )```

where

<next address>, INT,

is the 'G'[O] 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

4-18
example:

\[
@\text{next}\text{^-addr} := \text{NUMIN(in\text^-buffer, number, 10, stat)}; \\
\text{or} \\
\text{CALL NUMIN(in\text^-buffer, number, 10, status)};
\]

CONSIDERATIONS

• Number conversion stops on the first ASCII numerical character representing a value greater than \(<\text{base}>-1\) 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:

\[
\text{STRING number [0:9]} := "12345 "; \\
\text{INT result, status, .nextchar}
\]

Then NUMIN is invoked:

\[
@\text{next\textchar} := \text{NUMIN( number , result , 10 , status )} ; \\
\]

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

Then subtracting

\[
\text{num\textchar converted} := @\text{next\textchar} \textchar - \textchar @\text{number}; \\
\]

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

An alternate way of doing the same:

\[
\text{num\textchar converted} := \text{NUMIN(number,result,10,status) \textchar - \textchar @\text{number}} ; \\
\]

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

\[
\text{STRING number[0:5]} := "$\&19234" ; \\
\]

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:
UTILITY PROCEDURES
NUMOUT Procedure

"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
```
UTILITY PROCEDURES
TIME Procedure

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:

```plaintext
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);
```
UTILITY PROCEDURES
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:

\[
<\text{version}> := \text{TOSVERSION} \\
\text{-------} \\
\text{where} \\
<\text{version}>, \text{INT}, \\
\text{is returned a value of the form} \\
<0:7> \text{ upper-case ASCII letter indicating system level:} \\
\text{"A" = T.O.S.} \\
\text{"B" = GUARDIAN} \\
\text{"C" = GUARDIAN / 1.1} \\
\text{"D" = GUARDIAN / EXPAND} \\
\text{"E" = GUARDIAN / EXPAND / TMF} \\
\text{"K" = GUARDIAN, NonStop II system} \\
<8:15> \text{ revision number of system, in binary}
\]
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
CHECKPOINTING FACILITY

Introduction

- 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.
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.

Figure 5-1. A NonStop Program
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
- POSITION(f1,-1D); ! position to eof
- WRITE(f1,f1^buffer);
*** FAILURE OF PRIMARY ***
```

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

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 sent to its backup process.
CHECKPOINTING FACILITY
Checkpointing Procedures

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.

SETSNCINFO 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:
  - 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

```
\text{"G" \, [ $\text{MIN} \, ( \text{LASTADDR} , 32767 ) \, - \, 500 ]}
```
CHECKPOINTING FACILITY
CHECKMONITOR Procedure

- 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

<table>
<thead>
<tr>
<th>Primary</th>
<th>Backup (CHECKMONITOR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>No action</td>
<td>At beginning of CHECKMONITOR execution, the current state of cpu monitoring for the caller is saved (i.e., the current MONITORCPUS &lt;cpu mask&gt;), then MONITORCPUS is called, specifying only the primary's processor module.</td>
</tr>
<tr>
<td>CHECKOPEN</td>
<td>OPEN is called for the designated file.</td>
</tr>
<tr>
<td>CHECKPOINT</td>
<td>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.</td>
</tr>
<tr>
<td>CHECKCLOSE</td>
<td>CLOSE is called for the designated file.</td>
</tr>
<tr>
<td>CHECKSWITCH</td>
<td>First CHECKMONITOR calls RESETSINC 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.</td>
</tr>
<tr>
<td>Process Failure</td>
<td>(STOP or ABEND system message received for primary). First CHECKMONITOR calls RESETSINC 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.</td>
</tr>
</tbody>
</table>
Table 5-1. Action of CHECKMONITOR (cont'd)

<table>
<thead>
<tr>
<th>Primary</th>
<th>Backup (CHECKMONITOR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor Failure</td>
<td>(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.</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>CHECKOPEN parameter</th>
<th>corresponding OPEN parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;file name&gt;, INT:ref</td>
<td>= &lt;file name&gt;</td>
</tr>
<tr>
<td>&lt;file number&gt;, INT:value</td>
<td>= &lt;file number&gt;</td>
</tr>
<tr>
<td>&lt;flags&gt;, INT:value</td>
<td>= &lt;flags&gt;</td>
</tr>
<tr>
<td>&lt;sync or receive depth&gt;, INT:value</td>
<td>= &lt;sync or receive depth&gt;</td>
</tr>
<tr>
<td>&lt;sequential block buffer&gt;, INT:ref</td>
<td>= &lt;sequential block buffer&gt;</td>
</tr>
<tr>
<td>&lt;buffer length&gt;, INT:value</td>
<td>= &lt;buffer length&gt;</td>
</tr>
</tbody>
</table>

- The following parameter is required:

```
<back error>, INT:ref:l, >= 0, is the file management error number reflecting the call to OPEN in the backup process.
```
CHECKPOINTING FACILITY
CHECKOPEN Procedure

-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.
CHECKPOINTING FACILITY

CHECKPOINT Procedure

The CHECKPOINT procedure is called by a primary process to send information pertaining to its current executing state to its backup process. The 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.

5.2-12
<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.

eexample:

`@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
2, take over 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>[lst] = -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>[lst] = <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).
CHECKPOINTING FACILITY
CHECKPOINTMANY Procedure

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:

\[-\]

\(<0:7> = 1, \text{could not communicate with backup, then}\)
\(<0:7> = 2, <8:15> = 0 \text{primary stopped}\)
\(<0:7> = 2, <8:15> = 1 \text{primary abended}\)
\(<0:7> = 2, <8:15> = 2 \text{primary's processor failed}\)
\(<0:7> = 2, <8:15> = 3 \text{primary called CHECKSWITCH}\)

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.
CHECKPOINTING FACILITY
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, =
    
    \[ 7 + \text{longest alt key len} + \text{pri key len} + 1 \div 2 \]

- `<sync block size>`, INT:ref:l,
  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
    - ... (for modules 2 to 15)
    - `<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
Checkpointing Facility

MonitorCPUS Procedure

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:

  \[
  \text{<sysmsg>} = -3 \\
  \text{<sysmsg>[1]} = \text{<cpu>}
  \]

  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.

eample:

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

cpu^info := PROCESSORSTATUS;
```
CHECKPOINTING FACILITY
RESETPSYNC Procedure (disc and process files)

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

The RESETPSYNC 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 RESETPSYNC 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.

RESETPSYNC 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 RESETPSYNC procedure is:

```
CALL RESETPSYNC ( <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 RESETPSYNC was successful.
> (CCG) indicates that the file is not a disc file.
```

example:

```
CALL RESETPSYNC ( file^num );
IF < THEN ......; ! error
```
CHECKPOINTING FACILITY

SETSYNCINFO Procedure (disc and process files)

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:
      begin
      if created then
      begin
      8. open files in backup process
      9. use Awaitio to complete nowait opens in backup process
      end
      checkpoint environment to backup
      end
   end

5.3-1
Using the Checkpointing Facility

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

5.3-2
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 variables.
   rfnum, ! global data declarations.
   
   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.
   ! global data declarations.
   flags1 = ..., !
   sync^depth1 = ..., !
   flags2 = ..., !
   sync^depth2 = ..., !
   . !
   .
   flagsn = ..., !
   sync^depthn = ...; !

   INT .fname1[0:11], !
   fnum1, !
   .fname2[0:11], !
   fnum2, !
   .
   .
   .fname0[0:11], !
   fnumn; !
CHECKPOINTING FACILITY
Using the Checkpointing Facility

CALL OPEN ( fnamel, fnuml, flagsl, sync^depthl )
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;

5.3-4
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'>>'l0)'+ l, 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( fnamel,fnuml,flagsl,sync^depthl,,,error );
IF <> THEN ... ! see note.
CALL CHECKOPEN( fnamel2,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:
CHECKPOINTING FACILITY

Using the Checkpointing Facility

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 respond 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.
  .receive[0:11] := ["$RECEIVE", 8 * [" "]], !
  rfnum,
  stop^count := 0,
  backup^up := 0;

LITERAL
  flags1 = ..., !
  sync^depth1 = ..., !
  flags2 = ..., !
  sync^depth2 = ..., !
  ... !
  flagsn = ..., !
  sync^depthn = ...;

INT .fname[0:11],
  fnuml, !
  .fname2[0:11], !
  fnum2, !
  ...
  .fnum[0:11], !
  fnumn;

PROC m MAIN;
BEGIN
  INT .ppdentry[0:8], ! first local variable in MAIN.
Using the Checkpointing Facility

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(fnamel,fn numl,flagsl,sync^depthl)
  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;
The following is the example code in the "createbackup" procedure:

```plaintext
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( fnamel,fnuml,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 (,, fnuml,,, fnum2,,, .........,, fnumn );
                    CALL CHECKPOINT (, addr, count, ... );

                    IF (status := CHECKPOINT(stackbase)) THEN ! restart point.
                        CALL analyze^checkpoint^status ( status );
                    END; ! open files

                END; ! of createbackup

    END;
```

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.
CHECKPOINTING FACILITY
Using the Checkpointing Facility

- 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
      end
      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
      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.
CHECKPOINTING FACILITY
Using the Checkpointing Facility

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.
CHECKPOINTING FACILITY
Using the Checkpointing Facility

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:

5.3-12
CHECKPOINTING FACILITY
Using the Checkpointing Facility

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
CHECKPOINTING FACILITY
Using the Checkpointing Facility

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

For example, in the primary process:

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

INT .fnamel[0:ll],
    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.
CHECKPOINTING FACILITY
Using the Checkpointing Facility

\[
\begin{align*}
&\text{x READ a record} \\
&\text{x update it in memory} \\
&(c) \ \text{CHECKPOINT(stackbase,buffer,count,fnum);} \\
&\text{x WRITE a record}
\end{align*}
\]

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.

5.3-16
Example of Where Checkpoints Should Occur

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

The record:

```
| acctno | cur^bal | credit^limit |
```

The transaction cycle (without checkpoints):

```plaintext
WRITEREAD(terminal, buf1,..); ! returns <acctno> and <amount>.
POSITION (acctfile, acctno);
READUPDATE (acctfile, buf2,..);
IF (x := cur^bal + amount) > credit^limit THEN abort^transaction;
cur^bal := x;
WRITEUPDATE(acctfile, buf2,..);
WRITE (terminal, buf1,..); ! result
```
CHECKPOINTING FACILITY
Using the Checkpointing Facility

The transaction cycle with insufficient checkpoints:

1. CHECKPOINT(stk); idle state checkpoint.
   WRITEREAD(terminal, bufl,..); returns <acctno> and <amount>.

2. CHECKPOINT(stk, bufl, cnt); terminal data checkpoint.
   POSITION(acctfile, acctno);
   READUPDATE(acctfile, buf2,..);
   IF (x := cur^bal + amount) > credit^limit THEN abort^transaction;
   cur^bal := x;
   WRITEUPDATE(acctfile, buf2,..);
   WRITE(terminal, bufl,..); result

In this example, the first checkpoint identifies the program state as being idle (or waiting for 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,..);
CHECKPOINTING FACILITY
Using the Checkpointing Facility

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.
CHECKPOINTING FACILITY
Using the Checkpointing Facility

The transaction cycle with sufficient checkpointing:

1. **RESTART POINT**
   - **(1)** CHECKPOINT(stk); ! idle state checkpoint
   - x WRITEREAD(terminal, buf1, ..); ! returns <acctno> and <amount>.

2. **RESTART POINT**
   - **(2)** CHECKPOINT(stk, buf1, cnt); ! terminal data checkpoint.
   - x POSITION (acctfile, acctno);
   - x READUPDATE (acctfile, buf2, ..);
   - x IF (x := cur^bal + amount) > credit^limit THEN abort^transaction;
   - x cur^bal := x;

3. **RESTART POINT**
   - **(3)** CHECKPOINT(stk, buf2, cnt,.., acctfile); ! updated record checkpoint.
   - x WRITEUPDATE (acctfile, buf2, ..);
   - x WRITE (terminal, buf1, ..); ! result.

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.

5.3-20
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 \(<\text{sync depth}>\) parameter value of OPEN is specified as the maximum number of calls to WRITE for the file that are made between checkpoints for the file. Then, when a file is about to be updated by performing \(<\text{sync depth}>\) 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:

```plaintext
CALL READ ( rfnum, rbuffer, count ); ! no-wait.

CALL WRITE ( fnum1, 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,
CHECKPOINTING FACILITY

Using the Checkpointing Facility

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.
CHECKPOINTING FACILITY
Using the Checkpointing Facility

• 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:

\[
\begin{align*}
& \text{<sysmsg>} = -5 \\
& \text{<sysmsg>}[1] \text{ FOR } 4 = \text{ process ID of deleted process,}
\end{align*}
\]

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.

\[
\begin{align*}
& \text{<sysmsg>} = -5 \\
& \text{<sysmsg>}[1] \text{ FOR } 3 = \$<\text{process name}> \text{ of deleted process [pair]} \\
& \text{<sysmsg>}[4] = -1
\end{align*}
\]

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:

\[
\begin{align*}
& \text{<sysmsg>} = -6 \\
& \text{<sysmsg>}[1] \text{ FOR } 4 = \text{ process ID of deleted process}
\end{align*}
\]

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.

\[
\begin{align*}
& \text{<sysmsg>} = -6 \\
& \text{<sysmsg>}[1] \text{ FOR } 3 = \$<\text{process name}> \text{ of deleted process [pair]} \\
& \text{<sysmsg>}[4] = -1
\end{align*}
\]

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:
CHECKPOINTING FACILITY
Using the Checkpointing Facility

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
CHECKPOINTING FACILITY
Using the Checkpointing Facility

CHECKPOINTING FACILITY
Using the Checkpointing Facility

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:

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

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;
        END;
        ! 1.
        BEGIN ! 2 = takeover from primary.
            CASE status.<8:15> OF
                BEGIN ! 0 = primary stopped.
                    CALL STOP;
                    BEGIN
                        backup^up := 0;
                        stop^count := stop^count + 1;
                        CALL create^backup ( backup^cpu );
                    END;
                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;
    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.
```
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
            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
                    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.)

![Diagram of Backup Open by Backup Process]

**Figure 5-3. Backup Open by Backup Process**

NOTE: create_backup IS A DUMMY FUNCTION TO SIMULATE CREATION OF A BACKUP PROCESS BY THE PRIMARY PROCESS. "MOM" IS A PROCESS CONTROL PROCEDURE THAT RETURNS THE process ID OF A PROCESS'S CREATOR. send_msg AND read_msg ARE DUMMY FUNCTIONS TO SIMULATE SENDING AND RECEIVING INTERPROCESS MESSAGES.
CHECKPOINTING FACILITY
Advanced Checkpointing

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:
TAKEOVER
BY BACKUP

x SETSYNCINFO (fl, sync); x GETSYNCINFO(fl, sync);

(c) checkpoint sync block, position, and data.

x POSITION(fl,-1D); ! position to eof.

x WRITE(fl,fl^buffer,count);

*** FAILURE OF PRIMARY ***

(c) completion checkpoint

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

Another procedure, RESETSNC, 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 RESETSNC after taking over for the failed primary process, the backup process ensures that this does not occur.

A call to RESETSNC causes a RESETSNC 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 RESETSNC 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:

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

- Illegal Address Reference - an address was specified that was not within either the virtual code area or the virtual data area allocated to the process. Virtual code area allocation is determined by the size of the program's code area. By default, virtual data area allocation is determined by the TAL compiler to be equal to the number of memory pages needed for the program's global storage plus one memory page for the program's data stack. The size of the virtual data area can be increased 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
TRAPS AND TRAP HANDLING

Introduction

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.
TRAPS AND TRAP HANDLING

Introduction

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

\[ \$\text{MIN (LASTADDR,32767)} - 400 \]

- 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.
TRAPS AND TRAP HANDLING

ARMTRAP Procedure

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):
TRAPS AND TRAP HANDLING
ARMTRAP Procedure

'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>.
TRAPS AND TRAP HANDLING

ARMTRAP Procedure

<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

  \[
  \text{CODE( PUSH \#777 )}
  \]

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

  \[
  \text{CODE ( ADDS <num locals> )}
  \]

where \textquote{<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:

  \[
  \begin{array}{c|c}
  \text{trap} & \text{P register} \\
  \hline
  0 & I \\
  1 & I \\
  2 & I + 1 \\
  3 & ? \\
  4 & I \\
  11 & I \\
  12 & I \\
  13 & ? \\
  \end{array}
  \]

  where \(I\) = the address of the instruction being executed at the time of the trap.
  \(?\) = undefined.

- The \textquote{<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 \textquote{<trap label>} specified as \textquote{"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).
TRAPS AND TRAP HANDLING

ARMTRAP Procedure

- 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:

```plaintext
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.

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

formats and prints the message on the home terminal.
```
TRAPS AND TRAP HANDLING

ARMTRAP Procedure

    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.
SECURITY SYSTEM

Introduction

• 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>
```

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
MARKETING           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
```

<table>
<thead>
<tr>
<th>USER NAME</th>
<th>USER ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADMIN.BILL</td>
<td>1,2</td>
</tr>
</tbody>
</table>

Naming Conventions

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

<table>
<thead>
<tr>
<th>User Class</th>
<th>User Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>super ID</td>
<td>SUPER.SUPER</td>
</tr>
<tr>
<td>system operator</td>
<td>SUPER.OPERATOR</td>
</tr>
<tr>
<td>null</td>
<td>NULL.NULL</td>
</tr>
</tbody>
</table>

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").
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)
- **3** = owner (local); group ID and user ID match those of file’s owner
- **2** = member of owner’s group (local); group ID matches that of file’s owner
- **1** = any user (local)
- **0** = any user (local)
SECURITY SYSTEM
Introduction

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.

7.1-7
SECURITY SYSTEM
Introduction

Table 7-1. Allowability of File Access

<table>
<thead>
<tr>
<th>ACCESSOR'S SECURITY LEVEL</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>6</td>
<td>-</td>
<td>Y</td>
<td>Y</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>-</td>
<td>-</td>
<td>Y</td>
<td>Y</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>-</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>Y</td>
<td>Y</td>
<td>-</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>Y</td>
<td>-</td>
<td>-</td>
<td>Y</td>
<td>-</td>
</tr>
</tbody>
</table>

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.
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
SECURITY SYSTEM

Introduction

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

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

<table>
<thead>
<tr>
<th>Eligibility</th>
<th>Super ID User</th>
<th>Group Manager</th>
<th>Standard User</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>GIVE</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Changes ownership of file owned by current user.</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Changes ownership of any file.</td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td><strong>INFO</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lists a file's characteristics.</td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td><strong>LICENSE</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Permits privileged program to be run by users other than super ID.</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>REVOKE</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unlicenses privileged program.</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>SECURE</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sets file security of file owned by current user.</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Sets disc file security of any file.</td>
<td></td>
<td></td>
<td>x</td>
</tr>
</tbody>
</table>

## 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 |
SECURITY SYSTEM

Introduction

<table>
<thead>
<tr>
<th>Classification</th>
<th>Eligibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>Super ID User</td>
<td></td>
</tr>
<tr>
<td>Group Manager</td>
<td></td>
</tr>
<tr>
<td>Standard User</td>
<td></td>
</tr>
</tbody>
</table>

Process Control Procedures

**STEPMOM**  Changes creator process ID for designated process x x x if both processes have same process accessor ID.

Changes creator process ID for designated process regardless of process accessor ID’s.

**STOP**  Stops process started by creator, or that has same process accessor ID as creator, or that has stop mode of 0.

Stops any process.

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

```
REMTOPASSWORD <my system name>, <allow-access password>
```

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

```
REMTOPASSWORD <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
REMTOPASSWORD \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
REMTOPASSWORD \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 REMTOPASSWORD 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.
SECURITY SYSTEM
Introduction

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
SECURITY SYSTEM
CREATORACCESSID Procedure

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:

```plaintext
<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

eexample:
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 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.
SECURITY SYSTEM
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

<table>
<thead>
<tr>
<th>&lt;function&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 = set disc file security:</td>
</tr>
<tr>
<td>&lt;parameter 1&gt;</td>
</tr>
<tr>
<td>.&lt;0&gt; = 1, for program files only. Set accessor's ID to program file's ID when program file is run.</td>
</tr>
<tr>
<td>.&lt;1&gt;, clearonpurge file attribute; if set, clear data in the file before purging file directory.</td>
</tr>
<tr>
<td>.&lt;4:6&gt;, ID allowed for read.</td>
</tr>
<tr>
<td>.&lt;7:9&gt;, ID allowed for write.</td>
</tr>
<tr>
<td>.&lt;10:12&gt;, ID allowed for execute.</td>
</tr>
<tr>
<td>.&lt;13:15&gt;, ID allowed for purge.</td>
</tr>
</tbody>
</table>

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".
SECURITY SYSTEM
Functions for SETMODE and SETMODENOWAIT Procedures

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:

```plaintext
{ <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.
SECURITY SYSTEM
USERIDTOUSERNAME Procedure

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:

```plaintext
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 ... ;
```
SECURITY SYSTEM
VERIFYUSER Procedure

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
SECURITY SYSTEM
VERIFYUSER Procedure

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.
MEMORY MANAGEMENT PROCEDURES
Segmented Memory (NonStop II systems 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 PCB'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.
MEMORY MANAGEMENT PROCEDURES
Segmented Memory (NonStop II systems only)

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.
MEMORY MANAGEMENT PROCEDURES
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:

```lisp
status := ALLOCATESEGMENT ( segment^id, seg^size, swap^file );
  ! standard call to create a user segment;
  ! "swap^file" parameter may be omitted
```

```lisp
status := ALLOCATESEGMENT ( segment^id,,, pin );
  ! call to share the segment "segment^id"
  ! with the process "pin"
```
MEMORY MANAGEMENT PROCEDURES
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} );
MEMORY MANAGEMENT PROCEDURES

DEFINEPOOL Procedure (NonStop II systems only)

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:

```plaintext
<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 $1000000D$.

example:

```plaintext
status := DEFINEPOOL ( pool^head, pool, 2048 );
```
MEMORY MANAGEMENT PROCEDURES
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.

8.1-8
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.
MEMORY MANAGEMENT PROCEDURES
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:

\[
<\text{old segment id}> := \text{USESEGMENT} \left( <\text{segment id}> , <\text{pin}> \right)
\]

where

\[
<\text{old segment id}>, \text{INT},
\]

is returned the segment ID of the previously used segment, if any; otherwise, -1.

\[
<\text{segment id}>, \text{INT},
\]

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.

\[
<\text{pin}>, \text{INT},
\]

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:

\[
< \text{(CCL)} \]

indicates that \(<\text{segment id}>\) is not allocated, or that \(<\text{pin}>\) or an absolute segment was supplied by a non-privileged caller.

\[
= \text{(CCE)}
\]

indicates that the operation was successful.

\[
> \text{(CCG)}
\]

is not returned by USESEGMENT.

examples:

\[
\text{old}^{\text{seg}}\text{id} := \text{USESEGMENT} \left( \text{new}^{\text{seg}}\text{id} \right); \quad ! \text{change segments}
\]

\[
\text{segment}^{\text{id}} := \text{USESEGMENT}; \quad ! \text{get current segment ID}
\]

\[
\text{segment}^{\text{id}} := \text{USESEGMENT} \left( -1 \right); \quad ! \text{de-select extended segments}
\]

CONSIDERATIONS

- Because segment relocation is done, the first byte of any extended segment has the address $2000000D$.  

8.1-10
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.
MEMORY MANAGEMENT PROCEDURES
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> )

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.
MEMORY MANAGEMENT PROCEDURES
LOCKDATA Procedure (NonStop systems only)

- 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 processor 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):

   \[
   \text{<base page>} = \frac{(\text{<l limit>} + 1024)}{1024}
   \]

2. Take the <u base> address shown on the SYSGEN listing and divide that value by the page size in words:

   \[
   \text{<lim page>} = \frac{\text{<u base>}}{1024}
   \]

3. Subtract the <base page> value from the <lim page> value to obtain the number of available pages.

   \[
   \text{<num pages>} = \text{<lim page>} - \text{<base page>}
   \]

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
MEMORY MANAGEMENT PROCEDURES
LOCKDATA Procedure (NonStop systems only)

VIRTUAL SYSTEM DATA: 2 K
SYSTEM ADDRESS SPACE USED: %000000-%061777  %133463-%177777

<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:

\[
<\text{status}> := \text{LOCKMEMORY} ( <\text{address}> \\
\text{---} \text{---} \text{---} \\
, <\text{byte count}> \\
\text{---} \text{---} \text{---} \\
, <\text{timeout value}> \\
\text{---} \text{---} \text{---} \\
, <\text{parameter1}> \\
, <\text{parameter2}> ) \\
\]

where

\(<\text{status}>, \text{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.

\(<\text{address}>, \text{STRING:ref:EXT}, >

is the extended address of the block to be locked down.

\(<\text{byte count}>, \text{INT(32):value}, >

is the number of bytes to be locked.

\(<\text{timeout value}>, \text{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.
MEMORY MANAGEMENT PROCEDURES
LOCKMEMORY Procedure (NonStop II systems only)

<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
MEMORY MANAGEMENT PROCEDURES
LOCKMEMORY Procedure (NonStop II systems only)

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.)
MEMORY MANAGEMENT PROCEDURES

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.
SEQUENTIAL I/O PROCEDURES

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.
SEQUENTIAL I/O PROCEDURES
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 CHECKFILE procedure checks the file characteristics.

The call to the CHECKFILE procedure is:

\[
<\text{retval}> := \text{CHECKFILE} ( \{ \langle\text{common fcb}\rangle \}, \langle\text{operation}\rangle )
\]

where

\( <\text{retval}>, \text{INT}, \)

is the value returned for the requested operation.

\( <\text{common fcb}> \text{ or } <\text{file fcb}>, \text{INT:ref}, \)

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\(^\text{\textasciitilde}\)BREAKHIT, FILE\(^\text{\textasciitilde}\)ERRORFILE, and FILE\(^\text{\textasciitilde}\)TRACEBACK. Specifying an improper FCB causes an error indication.

\( <\text{operation}>, \text{INT:value}, \)

specifies which file characteristic is checked. The \( <\text{operation}>\text{s} \) and their associated \( <\text{ retval}>\text{s} \) are:

\( <\text{operation}> = \text{FILE}\(^\text{\textasciitilde}\)\text{ABORT\textasciitilde}XFERERR, \quad \text{(file must be open)} \)
\( <\text{retval}> := \langle\text{bit value}\rangle \)

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.

\( <\text{operation}> = \text{FILE}\(^\text{\textasciitilde}\)ASSIGNMASK1, \)
\( <\text{retval}> := \langle\text{high-order word of ASSIGN fieldmask}\rangle \)

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.

\( <\text{operation}> = \text{FILE}\(^\text{\textasciitilde}\)ASSIGNMASK2, \)
\( <\text{retval}> := \langle\text{low-order word of ASSIGN fieldmask}\rangle \)

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.
SEQUENTIAL I/O PROCEDURES
CHECKFILE 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,       (file must be open)
<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,      (file must be open)
< 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,
<retval> := <file number>

(file must be open)
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:

<table>
<thead>
<tr>
<th>byte numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0] [1] [8]</td>
</tr>
<tr>
<td>&lt;len&gt;&lt;logical file name&gt;</td>
</tr>
</tbody>
</table>

<len> is the length of the logical file name in bytes {0:7}.

<operation> = FILE^LOGIOOUT,
<retval> := <state of the logioout bit>

(file must be open)
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^RCVEEOF, (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.
SEQUENTIAL I/O PROCEDURES
CHECKAFILE Procedure

<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.
SEQUENTIAL I/O PROCEDURES
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.
SEQUENTIAL I/O PROCEDURES
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:

```verbatim
{ 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 OPENFILE procedure permits access to a file using the other sequential i/o procedures.

The call to the OPENFILE procedure is:

```
{ CALL }  OPENFILE ( <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 OPENFILE calls. The first word of the common FCB must be initialized to zero before the first OPENFILE 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 OPENFILE call is made.

See "Initializing the File FCB" following the description of the FCB structure.
OPENAFILE Procedure

<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 OPENAFILE, 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 OPENAFILE, 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\(^*\)TOP,

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.
OPEN FILE Procedure

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".
SEQUENTIAL I/O PROCEDURES
OPENFILE Procedure

CONSIDERATIONS

- If AUTO_TOKEN 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 OPENFILE to access a temporary disc file, AUTO_CREATE must be disabled; otherwise the OPENFILE call results in a file system error 13.

- All files opened with the OPENFILE 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 READAFILE procedure is used to read a file sequentially. The file must be open with read or read/write access.

The call to the READAFILE procedure is:

```
{ CALL := l READAFILE <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.
SEQUENTIAL I/O PROCEDURES
READ^FILE Procedure

<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>.

9-22
The SETFILE procedure alters file characteristics and checks the old value of those characteristics being altered.

The call to the SETFILE procedure is:

```
{ CALL { <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 SETFILE.

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 SETBREAKHIT, SETERRORFILE, and SETTRACEBACK. 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 SETFILE Operations".

<new value>, INT:value,

specifies a new value for the specified <operation>. This may be optional, depending on the operation desired.
SETAFILE Procedure

<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 SETAFILE OPERATIONS

This is a list of the file characteristics which can be altered by the SETAFILE procedure. All addresses passed are assumed to be integer addresses.

<operation> = ASSIGN^BLOCKBUFFLEN (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^PRIMARYEXXTENTSIZE)
<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."
SET^FILE Procedure

<operation> = ASSIGN^SECEXT (or, ASSIGN^SECONDARYEXTENTSIZEx)
<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        (optional)
<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).
SEQUENTIAL I/O PROCEDURES

SET^FILE Procedure

<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.

element:

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).

element:

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)
SEQUENTIAL I/O PROCEDURES

SET\FILE Procedure

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
<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
<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
<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
<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.
SEQUENTIAL I/O PROCEDURES

SET^FILE Procedure

<operation> = SET^SYSTEMMESSAGES
<new value> = <new sys-msg mask>
<old value> := <sys-msg mask>

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> = RESETSYSYNC 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
<new value> = @<new sys-msg mask words>
<old value> := <sys-msg mask words>

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.
SEQUENTIAL I/O PROCEDURES

SETFILE Procedure

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
SEQENIAL I/O PROCEDURES

SET^FILE Procedure

<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 PCB 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 );
```
SEQUENTIAL I/O PROCEDURES

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 \(<\text{time limit}>\) 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 \(<\text{time limit}>\) is 0D, then a check for completion is indicated. \text{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 \(<\text{error}>\) 40 and an "operation timed out" message is returned.

If \(<\text{time limit}>\) is 0D and \(<\text{error}>\) is 40, there is no completion. Therefore, \text{READ"FILE} or \text{WRITE"FILE} cannot be called for the file until the operation completes by \text{WAIT"FILE}. One method of determining if the operation completes is by the \text{CHECK"FILE} operation "FILE"LOGIOOUT". See "Checking File Transfer Characteristics".

element 1 - wait for completion:

\[
\text{CALL READ"FILE ( in^file , buffer , , , , l );}
\]
\[
\text{. }
\]
\[
\text{. }
\]
\[
\text{DO error := WAIT"FILE ( in^file , count )}
\]
\[
\text{UNTIL error <> SIOERR"IORESTARTED;}
\]

element 2 - check for completion:

\[
\text{IF NOT CHECK"FILE ( recv^file , FILE"LOGIOOUT ) THEN}
\]
\[
\text{CALL READ"FILE ( recv^file , recv^buf , , , , , l );}
\]
\[
\text{DO error := WAIT"FILE ( recv^file , recv^count , 0D )}
\]
\[
\text{UNTIL error <> SIOERR"IORESTARTED;}
\]
SEQUENTIAL I/O PROCEDURES
WRITE FILE Procedure

The WRITEFILE procedure writes a file sequentially. The file must be open with write or read/write access.

The call to the WRITEFILE procedure is:

```
{ CALL WRITEFILE ( <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.
WRITE^FILE Procedure

<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.

<form 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. <form control> corresponds to <parameter> of the file management CONTROL procedure for <operation> equal to 1. No forms control is performed if <form 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 );
SEQUENTIAL I/O PROCEDURES

Errors

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^INVALIDBUFSADDR
   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^BUFTOO_SMALL
   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^FILENAMEOPEN
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.
SEQUENTIAL I/O PROCEDURES

Errors

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](image-url)
SEQUENTIAL I/O PROCEDURES
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:

<table>
<thead>
<tr>
<th>Device</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operator</td>
<td>Write</td>
</tr>
<tr>
<td>Process</td>
<td>Read/Write</td>
</tr>
<tr>
<td>$RECEIVE</td>
<td>Read/Write</td>
</tr>
<tr>
<td>Disc</td>
<td>Read/Write</td>
</tr>
<tr>
<td>Terminal</td>
<td>Read/Write</td>
</tr>
<tr>
<td>Printer</td>
<td>Write</td>
</tr>
<tr>
<td>Mag Tape</td>
<td>Read/Write</td>
</tr>
<tr>
<td>Card Reader</td>
<td>Read</td>
</tr>
</tbody>
</table>

example:

```call
CALL SET^FILE ( in^file , ASSIGN^OPENACCESS , READ^ACCESS );
```

5. Specify exclusion for this open. This step is optional.

```call
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:

<table>
<thead>
<tr>
<th>Access</th>
<th>Exclusion Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read</td>
<td>if terminal then share, else protected</td>
</tr>
<tr>
<td>Write</td>
<td>if terminal then share, else exclusive</td>
</tr>
<tr>
<td>Read/Write</td>
<td>if terminal then share, else exclusive</td>
</tr>
</tbody>
</table>

example:

```call
CALL SET^FILE ( in^file , ASSIGN^OPENEXCLUSION , EXCLUSIVE );
```

6. Specify the logical record length. This step is optional.

```call
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:

<table>
<thead>
<tr>
<th>Device</th>
<th>Logical Record Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operator</td>
<td>132 bytes</td>
</tr>
<tr>
<td>Process</td>
<td>132 bytes</td>
</tr>
<tr>
<td>$RECEIVE</td>
<td>132 bytes</td>
</tr>
<tr>
<td>Unstructured Disc</td>
<td>132 bytes</td>
</tr>
<tr>
<td>Structured Disc</td>
<td>record length defined at creation</td>
</tr>
<tr>
<td>Terminal</td>
<td>132 bytes</td>
</tr>
<tr>
<td>Printer</td>
<td>132 bytes</td>
</tr>
<tr>
<td>Mag Tape</td>
<td>132 bytes</td>
</tr>
<tr>
<td>Card Reader</td>
<td>132 bytes</td>
</tr>
</tbody>
</table>

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 \(<\text{block length}>\) is specified, blocking is also specified. A physical block is composed of \(<\text{block length}>\) divided by \(<\text{record length}>\) logical records. When \(<\text{block length}>\) is not exactly divisible by \(<\text{record length}>\), 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 \(<\text{block length}>\) is specified. In the unspecified form, there is no indication of a physical block size; the records are contiguous on the medium.

CALL SETFILE ( \(<\text{file fcb}>\), ASSIGN^BLOCKLENGTH, \(<\text{block length}>\) )

\(<\text{block length}>\) is given in bytes.
SEQUENTIAL I/O PROCEDURES
Interface with INITIALIZER and Assign Messages

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.
SEQUENTIAL I/O PROCEDURES
Interface with INITIALIZER and Assign Messages

"#TEMP"

This means substitute a name appropriate for creating a temporary file for this name.

"#TEMP"

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:

```plaintext
INT .buf [ 0:11 ];
STRING .sbuf := @buf '<<' l;
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.

<table>
<thead>
<tr>
<th>CALL(S) TO SET FILE BEFORE CALLING INITIALIZER</th>
<th>CALL(S) TO SET FILE AFTER CALLING INITIALIZER</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPEN^FILE</td>
<td>OPEN^FILE</td>
</tr>
<tr>
<td>HIGHEST [ ASSIGN COMMAND ]</td>
<td>HIGHEST [ ASSIGN COMMAND ]</td>
</tr>
<tr>
<td>[ SET^FILE ]</td>
<td>[ SET^FILE ]</td>
</tr>
<tr>
<td>LOWEST DEFAULT SETTING</td>
<td>LOWEST DEFAULT SETTING</td>
</tr>
</tbody>
</table>

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
SEQUENTIAL I/O PROCEDURES
Interface with INITIALIZER and Assign Messages

follows (the FCB must be adjacent to other FCB's searched by the INITIALIZER):

\[
\text{INT .<file fcb> [ } 0:FCBSIZE - 1 ];
\]

In the executable part of the program, before calling the INITIALIZER, initialize the FCB:

\[
\text{CALL SET^FILE ( <file fcb> , INIT^FILEFCB );}
\]

Assign a logical file name, and any other open attributes desired, before calling the INITIALIZER:

\[
\text{CALL SET^FILE ( <file fcb> , ASSIGN^LOGICALFILENAME , @name );}
\]

\[
\text{CALL INITIALIZER ( .. );}
\]

\[
\text{CALL OPEN^FILE ( <common fcb> , <file fcb> , ... );}
\]

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.

9-50
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.
   .infilename,
   .ouftname,
   .device^type,
   phys^rec^len,
   interactive;

INT .buf [ 0:11 ];
STRING .sbuf := @buf `<<` 1; ! String corresponding to buf.

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.
   @infilename := CHECK^FILE( in^file, FILE^FILENAME^ADDR );
   @ouftname := CHECK^FILE( out^file, FILE^FILENAME^ADDR );

   ! Determine type of access for in file.
SEQUENTIAL I/O PROCEDURES
Interface with INITIALIZER and Assign Messages

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;
   @out^file := @in^file ! no need to
ELSE
   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;

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.
SEQUENTIAL I/O PROCEDURES
Usage Example Without INITIALIZER Procedure

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 - l],
.out^blkbuf [0:out^blkbuflen/2 - l];

?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
Usage Example Without INITIALIZER Procedure

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
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
  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;

. . .

9-55
SEQUENTIAL I/O PROCEDURES
NO^ERROR Procedure

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 := l
  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:

<table>
<thead>
<tr>
<th>Device</th>
<th>Retry Indication</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operator</td>
<td>yes</td>
</tr>
<tr>
<td>Process</td>
<td>n.a.</td>
</tr>
<tr>
<td>$RECEIVE</td>
<td>n.a.</td>
</tr>
<tr>
<td>Disc</td>
<td>(opened with sync depth of 1, so n.a.)</td>
</tr>
<tr>
<td>Terminal</td>
<td>yes</td>
</tr>
<tr>
<td>Printer</td>
<td>yes</td>
</tr>
<tr>
<td>Mag Tape</td>
<td>no</td>
</tr>
<tr>
<td>Card Reader</td>
<td>no</td>
</tr>
</tbody>
</table>

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 ] := [ l, 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 );
```
SEQUENTIAL I/O PROCEDURES

ERROR HANDLING BY NOERROR Procedure

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
if (retrycount := retrycount + 1) > 1 then
    goto fatal
else return retry indication.

if not retryable or
    (retrycount := retrycount + 1) > 1 then
    goto fatal
else
    if device <> mag tape and
        device <> card reader then
        return retry indication
    else
        goto fatal

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

$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

$RECEIVE = 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.
SEQUENTIAL I/O PROCEDURES
$RECEIVE Handling

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.
        .
        .
        fnnum := -1;
        CALL AWAITIO ( fnnum , , countread , , 3000D ) ;
        IF fnnum = 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. @filename instead of filename. The value returned by CHECK^FILE is the address of the item.
<table>
<thead>
<tr>
<th>Sequence</th>
<th>SET^FILE</th>
<th>CHECK^FILE</th>
</tr>
</thead>
<tbody>
<tr>
<td>C -</td>
<td>Assign^blockbuflen</td>
<td>A File^blockbuflen</td>
</tr>
<tr>
<td>C -</td>
<td>filecode</td>
<td>A filecode</td>
</tr>
<tr>
<td>C</td>
<td>filename</td>
<td>A ^filename^addr</td>
</tr>
<tr>
<td>C</td>
<td>logicalfilename</td>
<td>A logicalfilename^addr</td>
</tr>
<tr>
<td>C -</td>
<td>openaccess</td>
<td>A openaccess</td>
</tr>
<tr>
<td>C -</td>
<td>openexclusion</td>
<td>A openexclusion</td>
</tr>
<tr>
<td>C</td>
<td>priext</td>
<td>A priext</td>
</tr>
<tr>
<td>C -</td>
<td>recordlen</td>
<td>A recordlen</td>
</tr>
<tr>
<td>C -</td>
<td>seext</td>
<td>A seext</td>
</tr>
<tr>
<td>C o +</td>
<td>Init^filefcb [1]</td>
<td>o File^abort^xfererr</td>
</tr>
<tr>
<td>C</td>
<td>Set^abort^xfererr</td>
<td>A breakhit</td>
</tr>
<tr>
<td>A A</td>
<td>checksum</td>
<td>A checksum</td>
</tr>
<tr>
<td>o +</td>
<td>countxferred</td>
<td>o countxferred</td>
</tr>
<tr>
<td>o +</td>
<td>crlf^break</td>
<td>o * dupfile</td>
</tr>
<tr>
<td>o -</td>
<td>error</td>
<td>o errorfile</td>
</tr>
<tr>
<td>o -</td>
<td>errorfile</td>
<td>CA * errorfile</td>
</tr>
<tr>
<td>C o -</td>
<td>logioout</td>
<td>o logioout</td>
</tr>
<tr>
<td>R o</td>
<td>openerspid</td>
<td>CA * openerspid^addr</td>
</tr>
<tr>
<td>o +</td>
<td>physioout</td>
<td>o physioout</td>
</tr>
<tr>
<td>o +</td>
<td>print^err^msg</td>
<td>o print^err^msg</td>
</tr>
<tr>
<td>o +</td>
<td>prompt</td>
<td>o prompt</td>
</tr>
<tr>
<td>R o</td>
<td>rcveof</td>
<td>R o rcveof</td>
</tr>
<tr>
<td>R o</td>
<td>rcvopencnt</td>
<td>R o rcvopencnt</td>
</tr>
<tr>
<td>R o</td>
<td>rcvuseropenreply</td>
<td>R o rcvuseropenreply</td>
</tr>
<tr>
<td>R o</td>
<td>systemmessages</td>
<td>R o systemmessages</td>
</tr>
<tr>
<td>R o</td>
<td>systemmessagemany</td>
<td>R o * systemmessagemany</td>
</tr>
<tr>
<td>C A</td>
<td>traceback</td>
<td>CA * traceback</td>
</tr>
<tr>
<td>o +</td>
<td>userflag</td>
<td>o userflag</td>
</tr>
<tr>
<td>o +</td>
<td>write^fold</td>
<td>o write^fold</td>
</tr>
<tr>
<td>o +</td>
<td>write^pad</td>
<td>o write^pad</td>
</tr>
<tr>
<td>o +</td>
<td>write^trim</td>
<td>o write^trim</td>
</tr>
</tbody>
</table>

[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:

```
{ status := } FORMATCONVERT ( <iformal >
CALL ---------------------
, <iformal len>
, <eformal >
, <eformal len>
, <scales>
, <scalecount>
, <conversion> )
```

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 (`<iformal>`).
  - If = 0, indicates `<iformal len>` 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 `<eformal>` is numbered 1.

- `<iformal>, 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 `<iformal>` must be aligned on a word boundary, or the contents of `<iformal>` 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.

<iformatlen>, 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.
formatter
formatconvert procedure

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:

```plaintext
{ <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:

<table>
<thead>
<tr>
<th>WORD</th>
<th>CONTENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td>dataptr</td>
</tr>
<tr>
<td>[1]</td>
<td>datatype</td>
</tr>
<tr>
<td>[2]</td>
<td>databytes</td>
</tr>
<tr>
<td>[3]</td>
<td>dataoccurs</td>
</tr>
<tr>
<td>[4]</td>
<td>nullptr  (OPTIONAL)</td>
</tr>
</tbody>
</table>

<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.
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 FORMATLEN = 60 ;
STRING .FORMAT [0:FORMATLEN] :=
  "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:FORMATLEN/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] := [ lD, lD, 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
FORMATER

Example

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

10-12
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".
FORMATTER
Format Characteristics

**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:

<table>
<thead>
<tr>
<th>format:</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ fmt-item } [ separator fmt-item ]</td>
</tr>
<tr>
<td>{ b-separator } [ [separator] b-separator ]</td>
</tr>
</tbody>
</table>

fmt-item: { non-repeatable-edit-descriptor }

field-group: [repeat] [mods] { group-spec }

group-spec: { repeatable-edit-descriptor }

"(" format ")"

repeat: an unsigned, non-zero integer

mods: "[" { modifier } , . . . "]"
FORMATTER

Format Characteristics

**separators:** { , | / | : }

**b-separators:** { / | : }

<table>
<thead>
<tr>
<th>non-repeatable edit descriptors</th>
<th>repeatable edit descriptors</th>
<th>modifiers</th>
<th>decorations</th>
</tr>
</thead>
<tbody>
<tr>
<td>BN SP A G BN OC P</td>
<td>BZ SS D I BZ RJ M A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H T E L FL SS Z F</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>string TL F M LJ... F</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P TR x</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Some sample formats:

1. \( I5,F10.2 \)
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.
FORMATTER
Format Characteristics

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
Formatter
Edit Descriptors

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:
Edit Descriptors

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:

<table>
<thead>
<tr>
<th>Tn</th>
<th>TLn</th>
<th>TRn</th>
<th>nX</th>
</tr>
</thead>
</table>

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"

<table>
<thead>
<tr>
<th>Format</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>No tabs</td>
<td>I3,E12.4,A5</td>
</tr>
<tr>
<td></td>
<td>100 0.1000E+04HELLO</td>
</tr>
<tr>
<td></td>
<td>\ / \ / \ / \ /</td>
</tr>
<tr>
<td>X</td>
<td>I3,E12.4,1X,A5</td>
</tr>
<tr>
<td></td>
<td>100 0.1000E+04 HELLO</td>
</tr>
<tr>
<td></td>
<td>\ / \ / \ / \ /</td>
</tr>
<tr>
<td>TL</td>
<td>I3,E12.4,TL3,A5</td>
</tr>
<tr>
<td></td>
<td>100 0.1000EHELLO</td>
</tr>
<tr>
<td></td>
<td>\ / \ / \ / \ /</td>
</tr>
<tr>
<td>TR</td>
<td>I3,E12.4,TR5,A5</td>
</tr>
<tr>
<td></td>
<td>100 0.1000E+04 HELLO</td>
</tr>
<tr>
<td></td>
<td>\ / \ / \ / \ /</td>
</tr>
<tr>
<td>T</td>
<td>I3,E12.4,T3,A5</td>
</tr>
<tr>
<td></td>
<td>10HELLO1000E+04</td>
</tr>
<tr>
<td></td>
<td>\ / \ / \ / \ /</td>
</tr>
</tbody>
</table>

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:

\[ dc \ldots c d \quad OR \quad nhc \ldots c \]

\[ 1 2 3 \ldots n \quad OR \quad 1 2 3 \ldots n \]

where

\[ d = \text{either an apostrophe ('') or a quotation mark ('"'); the same character must be used for both the opening and closing delimiters.} \]

\[ c = \text{any ASCII character.} \]

\[ n = \text{an unsigned nonzero integer constant specifying the number of characters in the string; "n" may not exceed 255.} \]
Non-Repeatable Edit Descriptors

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:

<table>
<thead>
<tr>
<th>S</th>
<th>SP</th>
<th>SS</th>
</tr>
</thead>
</table>

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.
FORMATTER
Non-Repeatable Edit Descriptors

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.
Non-Repeatable Edit Descriptors

- 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

```
10(\' NUMBER \',I1,:/
```

Results:

```
| NUMBER 1 |
| NUMBER 2 |
| NUMBER 3 |
```

Without colon

```
10(\' NUMBER \',I1,/
```

Results:

```
| NUMBER 1 |
| NUMBER 2 |
| NUMBER 3 |
```

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:

\[
\begin{align*}
Aw & \quad \text{OR} \quad A \\
\text{where} \\
\ w & = \text{an unsigned integer constant that specifies the width, in characters, of the field and may not exceed 255.} \\
& \text{The field processed is the next } w \text{ characters starting at the current position.} \\
& \text{If } w \text{ is not present, the field width is equal to the actual number of bytes in the associated data element.} \\
& \text{After the field is processed, the current position is advanced by } w \text{ characters.}
\end{align*}
\]

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(l6) 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:

<table>
<thead>
<tr>
<th>Format</th>
<th>Data Value</th>
<th>External Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>'WORD'</td>
<td></td>
</tr>
<tr>
<td>A4</td>
<td>'WORD'</td>
<td></td>
</tr>
<tr>
<td>A3</td>
<td>'WORD'</td>
<td>WOR (overflow set)</td>
</tr>
<tr>
<td>[RJ]A3</td>
<td>'WORD'</td>
<td>ORD (overflow set)</td>
</tr>
<tr>
<td>A5</td>
<td>'WORD'</td>
<td>WORD</td>
</tr>
<tr>
<td>[RJ]A5</td>
<td>'WORD'</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>%041111</td>
<td>HI</td>
</tr>
</tbody>
</table>

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.
Formatter
Repeatable Edit Descriptors

Examples:

<table>
<thead>
<tr>
<th>External Field</th>
<th>Format</th>
<th>Data Item Length</th>
<th>Data Element Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>HELLO</td>
<td>A5</td>
<td>5 characters</td>
<td>'HELLO'</td>
</tr>
<tr>
<td>HELLO</td>
<td>A3</td>
<td>3 characters</td>
<td>'HEL'</td>
</tr>
<tr>
<td>HELLO</td>
<td>A6</td>
<td>6 characters</td>
<td>'HELLO'</td>
</tr>
<tr>
<td>HELLO</td>
<td>A5</td>
<td>6 characters</td>
<td>'HELLO'</td>
</tr>
</tbody>
</table>

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:

\[ \text{Dw.d} \]

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:

\[ Ew.d \quad \text{OR} \quad Ew.dEe \]

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".
Repeatable Edit Descriptors

On output, the field (for a scale factor of zero) appears in the following form:

\[
\{ [+\} \ [0]. n \ n \ ... \ n \ E \{+\} \ e \ e \ ... \ e \\
\{ -\} \ \ 1 \ 2 \ d \ \ \{ -\} \ 1 \ 2 \ e
\]

where

\[
\{ [+\}
\]

indicates an optional plus or a minus.

\[
\{ -\}
\]

n n ... n are the "d" most significant digits of the value of the datum after rounding.

\[
E
\]
signals the start of the decimal exponent.

\[
\{ +\}
\]

indicates that a plus or minus is required.

\[
\{ -\}
\]

e 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:

<table>
<thead>
<tr>
<th>Format</th>
<th>Data Value</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>El2.3</td>
<td>8.76543 x 10</td>
<td>0.877E-05</td>
</tr>
<tr>
<td>El2.3</td>
<td>-0.55555</td>
<td>-0.556E+00</td>
</tr>
<tr>
<td>El2.3</td>
<td>123.4567</td>
<td>0.123E+03</td>
</tr>
<tr>
<td>El2.6E1</td>
<td>3.14159</td>
<td>0.314159E+1</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>External Field</th>
<th>Format</th>
<th>Data Element Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.100E+03</td>
<td>El2.3</td>
<td>100</td>
</tr>
<tr>
<td>100.05</td>
<td>El2.5</td>
<td>100.05</td>
</tr>
<tr>
<td>12345</td>
<td>El2.3</td>
<td>12.345</td>
</tr>
</tbody>
</table>

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:

\[ Fw.d \quad \text{OR} \quad Fw.d.m \]

where

\[ w = \text{an unsigned integer constant that defines the total field width and may not exceed 255. The field processed is the "} \]

\[ w \text{" characters starting at the current position. After the field is processed, the current position is advanced by "} \]

\[ w \text{" characters.} \]

\[ d = \text{an unsigned integer constant that defines the number of digits that are to appear to the right of the decimal point in the external field.} \]
formatter
repeatable edit descriptors

\[ m = \text{an unsigned integer constant that defines the number of digits that must be present to the left of the decimal point on input.} \]

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:

<table>
<thead>
<tr>
<th>Format</th>
<th>Data Value</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>F10.4</td>
<td>123.4567</td>
<td>123.4567</td>
</tr>
<tr>
<td>F10.4</td>
<td>0.000123</td>
<td>0.0001</td>
</tr>
<tr>
<td>F10.4.3</td>
<td>-4.56789</td>
<td>-004.5679</td>
</tr>
</tbody>
</table>

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:

\[
\begin{align*}
Gw.d & \quad \text{OR} \quad Gw.dEe \\
\end{align*}
\]

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:

<table>
<thead>
<tr>
<th>Magnitude of Data</th>
<th>Equivalent Conversion Effected</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not Less Than</td>
<td>Less Than</td>
</tr>
<tr>
<td>0.1</td>
<td>0.1</td>
</tr>
<tr>
<td>0.1</td>
<td>1.0</td>
</tr>
<tr>
<td>1.0</td>
<td>10.0</td>
</tr>
<tr>
<td>10.0</td>
<td>100.0</td>
</tr>
<tr>
<td>( \cdot )</td>
<td>( \cdot )</td>
</tr>
<tr>
<td>( \cdot )</td>
<td>( \cdot )</td>
</tr>
<tr>
<td>( 10^{(d-2)} )</td>
<td>( 10^{(d-1)} )</td>
</tr>
<tr>
<td>( 10^{d} )</td>
<td>( 10^{d} )</td>
</tr>
<tr>
<td>( Ew.d )</td>
<td>( Ew.d )</td>
</tr>
</tbody>
</table>

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:
Formatter
Repeatable Edit Descriptors

<table>
<thead>
<tr>
<th>Value</th>
<th>&quot;F&quot; Fl3.6 Conversion</th>
<th>&quot;G&quot; Gl3.6 Conversion</th>
</tr>
</thead>
<tbody>
<tr>
<td>.01234567</td>
<td>0.012346</td>
<td>0.123457E-01</td>
</tr>
<tr>
<td>.12345678</td>
<td>0.123457</td>
<td>0.123457</td>
</tr>
<tr>
<td>1.2345678</td>
<td>1.234568</td>
<td>1.23457</td>
</tr>
<tr>
<td>12.34567890</td>
<td>12.345679</td>
<td>12.345</td>
</tr>
<tr>
<td>123.45678900</td>
<td>123.456789</td>
<td>123.457</td>
</tr>
<tr>
<td>1234.56789000</td>
<td>1234.567890</td>
<td>1234.57</td>
</tr>
<tr>
<td>12345.67890000</td>
<td>12345.678900</td>
<td>12345.7</td>
</tr>
<tr>
<td>123456.78900000</td>
<td>123456.789000</td>
<td>12345.7</td>
</tr>
<tr>
<td>1234567.89000000</td>
<td>****************</td>
<td>0.123457E+07</td>
</tr>
</tbody>
</table>

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.

10-34
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:

<table>
<thead>
<tr>
<th>Format</th>
<th>Data Value</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>I7</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>I7.2</td>
<td>-1</td>
<td>-01</td>
</tr>
<tr>
<td>I7.6</td>
<td>100</td>
<td>000100</td>
</tr>
<tr>
<td>I7.6</td>
<td>-1</td>
<td>-000001</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>External Field</th>
<th>Format</th>
<th>Data Element Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>I7</td>
<td>100</td>
</tr>
<tr>
<td>-01</td>
<td>I7</td>
<td>-1</td>
</tr>
<tr>
<td>1</td>
<td>I7</td>
<td>1</td>
</tr>
<tr>
<td>1 1000</td>
<td>BZ,I7</td>
<td>1000</td>
</tr>
<tr>
<td>1 10200</td>
<td>BZ,I7</td>
<td>10200</td>
</tr>
<tr>
<td>1 12</td>
<td>BN,I7</td>
<td>12</td>
</tr>
</tbody>
</table>

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.
Repeatable Edit Descriptors

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:

<table>
<thead>
<tr>
<th>Format</th>
<th>Data Value</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>L2</td>
<td>-1</td>
<td>T</td>
</tr>
<tr>
<td>L2</td>
<td>15769</td>
<td>T</td>
</tr>
<tr>
<td>L2</td>
<td>0</td>
<td>F</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>External Field</th>
<th>Format</th>
<th>Data Element Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>L7</td>
<td>-1</td>
</tr>
<tr>
<td>F</td>
<td>L7</td>
<td>0</td>
</tr>
<tr>
<td>.TRUE.</td>
<td>L7</td>
<td>-1</td>
</tr>
<tr>
<td>.FALSE.</td>
<td>L7</td>
<td>0</td>
</tr>
<tr>
<td>TUGBOAT</td>
<td>L7</td>
<td>-1</td>
</tr>
<tr>
<td>FARLEY</td>
<td>L7</td>
<td>0</td>
</tr>
</tbody>
</table>

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:

\[ M<\text{mask}> \]

where

\[ <\text{mask}> = \text{a character string enclosed in a pair of apostrophes ('), a pair of quotation marks ("), or less-than and greater-than symbols (<>). The string supplied must not exceed 255.} \]

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.
FORMATTER
Repeatable Edit Descriptors

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:

<table>
<thead>
<tr>
<th>Format</th>
<th>Data Value</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>M&quot;99/99/99&quot;</td>
<td>103179</td>
<td>10/31/79</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Format</th>
<th>Data Values</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>3M&lt;Z99.99&gt;</td>
<td>-27.40, 12, 0</td>
<td>-27.40 12.00 00.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td>\ \ \</td>
</tr>
<tr>
<td>3M&lt;ZZ9.99&gt;</td>
<td>-27.40, 12, 0</td>
<td>-27.40 12.00 0.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td>\ \ \</td>
</tr>
<tr>
<td>3M&lt;ZZZ.99&gt;</td>
<td>-27.40, 12, 0</td>
<td>-27.40 12.00 00</td>
</tr>
<tr>
<td></td>
<td></td>
<td>\ \ \</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Format</th>
<th>Data Value</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>M'Z,ZZ9.99'</td>
<td>32.009</td>
<td>32.01</td>
</tr>
</tbody>
</table>

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>

<table>
<thead>
<tr>
<th>Format One</th>
<th>Format Two</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ 298,738,472 AND NO CENTS</td>
<td>$ 298,738,472 AND NO CENTS</td>
</tr>
<tr>
<td>$ 389,488 AND NO CENTS</td>
<td>$ 000,389,488 AND NO CENTS</td>
</tr>
<tr>
<td>$ 666 AND NO CENTS</td>
<td>$ 000,000,666 AND NO CENTS</td>
</tr>
<tr>
<td>$ 0 AND NO CENTS</td>
<td>$ 000,000,000 AND NO CENTS</td>
</tr>
</tbody>
</table>

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:

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$ 9,758</td>
<td>3/17/77</td>
<td>WEST</td>
<td>(213) 529-6800</td>
<td></td>
</tr>
<tr>
<td>$21,573</td>
<td>9/17/79</td>
<td>MIDWEST</td>
<td>(216) 229-6270</td>
<td></td>
</tr>
<tr>
<td>$15,532</td>
<td>9/5/79</td>
<td>SOUTH</td>
<td>(404) 729-8400</td>
<td></td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>Format</th>
<th>Data Value</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>[FL'.']A10</td>
<td>'THEN'</td>
<td>THEN........</td>
</tr>
<tr>
<td>[RJ,FL&quot;&gt;&quot;]A10</td>
<td>'HERE'</td>
<td>&gt;&gt;&gt;&gt;&gt;&gt;HERE</td>
</tr>
<tr>
<td>[FL&quot;*&quot;]M&lt;ZZ,ZZ9.99&gt;</td>
<td>127.39</td>
<td>$***127.39</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Format</th>
<th>Data Value</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>[OC'!']12</td>
<td>100</td>
<td>!!</td>
</tr>
</tbody>
</table>

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.
FORMATTER
Modifiers

The justification modifiers are:

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LJ</td>
<td>Left justify (normal)</td>
</tr>
<tr>
<td>RJ</td>
<td>Right justify (data is displayed right-justified)</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>Digit selector (&quot;M&quot; format)</td>
</tr>
<tr>
<td>Z</td>
<td>Digit selector, zero suppression (&quot;M&quot; format)</td>
</tr>
<tr>
<td>V</td>
<td>Decimal alignment character (&quot;M&quot; format)</td>
</tr>
<tr>
<td>.</td>
<td>Decimal point (&quot;D&quot;, &quot;E&quot;, &quot;F&quot;, &quot;G&quot;, and &quot;M&quot; format)</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>Data Value</th>
<th>Format</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>12.45</td>
<td>[SS&quot;.:&quot;]F6.2</td>
<td>12:45</td>
</tr>
<tr>
<td>12.45</td>
<td>[SS&quot;.:&quot;]M<a href="">ZZZ:99</a></td>
<td>12:45</td>
</tr>
<tr>
<td>12345.67</td>
<td>[SS&quot;.,&quot;]F10.2</td>
<td>12345,67</td>
</tr>
<tr>
<td>103179</td>
<td>[SS&lt;9X&gt;]M&lt;XX/XX/19XX&gt;</td>
<td>10/31/1979</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>EDIT DESCRIPTORS</th>
<th>A</th>
<th>E,D</th>
<th>F</th>
<th>G</th>
<th>I</th>
<th>L</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>BZ,BN</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>LJ,RJ</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OC</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>FL</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>SS</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
</tr>
</tbody>
</table>
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 } ... { F } <string> OR { M }
{ N } { P } ...
{ P } ...
{ Z } { P } ...
```

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.

- \(<\text{string}>\) = 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:

<table>
<thead>
<tr>
<th>Format</th>
<th>Data Value</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>[MF&lt;,MP&gt;,ZPP]F12.2</td>
<td>1000.00</td>
<td>1,000.00</td>
</tr>
<tr>
<td>[MF&lt;,MP&gt;,ZPP]F12.2</td>
<td>-1000.00</td>
<td>&lt;1,000.00&gt;</td>
</tr>
<tr>
<td>[MAL,CR,MPF$]F12.2</td>
<td>1000.00</td>
<td>$1,000.00</td>
</tr>
<tr>
<td>[MAL,CR,MPF$]F12.2</td>
<td>-100.00</td>
<td>CR $100.00</td>
</tr>
<tr>
<td>[OAl&lt;<strong>OVERFLOW</strong>&gt;]F12.2</td>
<td>10000000.00</td>
<td>1,000,000.00</td>
</tr>
<tr>
<td>[OAl&lt;<strong>OVERFLOW</strong>&gt;]F12.2</td>
<td>10000000.00</td>
<td><strong>OVERFLOW</strong></td>
</tr>
</tbody>
</table>

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'-'
- OAl'*** ... *' (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 "OAl'***...*" 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

\[
\text{Format} \\
\text{[MP'-'\{F10.2,\[MZF'**'\}F10.2]} \\
\text{[MF'-'\}F10.2,\[MZF'**',MF'-'\}F10.2}
\]

Using the format above:

<table>
<thead>
<tr>
<th>Data Values</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>0,0</td>
<td>0.00 **0.00</td>
</tr>
<tr>
<td></td>
<td>(\backslash) (\backslash) (\backslash)</td>
</tr>
<tr>
<td>1,1</td>
<td>1.00  1.00</td>
</tr>
<tr>
<td></td>
<td>(\backslash) (\backslash) (\backslash)</td>
</tr>
<tr>
<td>-1,-1</td>
<td>-1.00 **-1.00</td>
</tr>
<tr>
<td></td>
<td>(\backslash) (\backslash) (\backslash)</td>
</tr>
</tbody>
</table>

Note: The "/\" marker is used to denote the boundaries of the output field.
FORMATTER
List-Directed Formatting

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:

\[
\begin{align*}
    r^c & \quad r^* \\
    \text{where} & \\
    r & \text{ is an unsigned, nonzero, integer constant.} \\
    r^c & \text{ form is equivalent to } r \text{ successive appearances of the constant } "c". \\
    r^* & \text{ form is equivalent to } r \text{ successive null values.} \\
    \text{Neither of these forms may contain embedded blanks, except where permitted with the constant } "c". \\
\end{align*}
\]

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
COMMAND INTERPRETER/APPLICATION INTERFACE
General Characteristics

- 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:

    $STOREL.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:

\[
$<\text{process name}> [. #<\text{lst qualif name}> [. <\text{2nd qualif name}> ] ]
\]

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](image)

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:
The Command Interpreter changes the logon default entry in the USERID file to "$VOL1.ACCTS". Note that the Command Interpreter's current default setting is unaffected by the DEFAULT command; the new logon default setting is not used until the next logon.

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:
COMMAND INTERPRETER/APPLICATION INTERFACE

File Names

:VOLUME bill

<table>
<thead>
<tr>
<th>$VOLL</th>
<th>BILL</th>
</tr>
</thead>
</table>

CURRENT DEFAULT SETTING

:EDIT myfile

myfile = $VOLL.BILL.MYFILE

T9601B04 EDIT (780113)
CURRENT FILE IS $VOLL.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

<table>
<thead>
<tr>
<th>Original File Name</th>
<th>Expanded File Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>myfile</td>
<td>\calif.$myvol.mysubvol.myfile</td>
</tr>
<tr>
<td>\newyork.myfile</td>
<td>\newyork.$myvol.mysubvol.myfile</td>
</tr>
<tr>
<td>$proc</td>
<td>\calif.$proc</td>
</tr>
</tbody>
</table>

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).
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.
COMMAND INTERPRETER/APPLICATION INTERFACE

RUN Command

example:

:VOLUME $storel.acctrctr
:RUN xnstp /IN infile,OUT outfile,NAME/ 1,10,byte,descending
! runs "$storel.acctrctr.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:

```c
STRUCT ciAstartup;
BEGIN
  INT msgcode;
  ! [0] -l.
  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; ! ciAstartupAmsg.
```

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:

\[
\begin{align*}
\text{word}[0] &= -1 \quad \text{! means start-up message.} \\
\text{word}[1] &= "\$STORE1 " \quad \text{! $<\text{default volume name}>$.} \\
\text{word}[5] &= \"ACCTRCV \" \quad \text{! <default subvol name>.} \\
\text{word}[9] &= "\$STORE1 ACCTRCV INFILE \" \quad \text{! IN param <file name>.} \\
\text{word}[21] &= "\$STORE1 ACCTRCV OUTFILE \" \quad \text{! OUT param <file name>.} \\
\text{word}[33] &= "1,10,byte,descending" \quad \text{! <parameter string>.} \\
\text{word}[43] &= <null><null> \quad \text{null terminator(s).}
\end{align*}
\]

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;
```
COMMAND INTERPRETER/APPLICATION INTERFACE

Startup Message

\[
\text{CALL OPEN(}\text{receive, recv}^\text{fnum}); \\
\text{CALL READ(}\text{recv}^\text{fnum, buffer, 200, num}^\text{read}); \\
\]

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

\[
\text{IF buffer <> -1 THEN CALL ABEND}; \\
\]

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

\[
\text{CALL OPEN(buffer[9], in}^\text{fnum}); \\
\text{CALL OPEN(buffer[21], out}^\text{fnum}); \\
\]

then saves the <parameter string> information:

\[
\text{num := num}^\text{read} - 66; ! length of parameter string in bytes.} \\
\text{IF num <= 40 THEN } ! \text{parameter string will fit. move it in.} \\
\text{parms := sbuffer[66] FOR num} \\
\text{ELSE } ! \text{parameter string too long.} \\
\]
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> ] ] ]
```

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.
COMMAND INTERPRETER/APPLICATION INTERFACE
ASSIGN Command

<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:

\[
\begin{align*}
\{ & \text{<extent spec>} \\
& \text{CODE <file code>} \\
& \text{<exclusion spec>} \\
& \text{<access spec>} \\
& \text{REC <record size>} \\
& \text{BLOCK <block size>} \\
\}
\end{align*}
\]

<extent spec> is

\[
\begin{align*}
\{ & \text{EXT [ ( ] <pri extent size> [ ] ) ]} \\
& \text{EXT ( [ <pri extent size > , <sec extent size> ) ]}
\}
\end{align*}
\]

<exclusion spec> is

\[
\begin{align*}
\{ & \text{EXCLUSIVE} \\
& \text{SHARED} \\
& \text{PROTECTED}
\}
\end{align*}
\]

<access spec>

\[
\begin{align*}
\{ & \text{I-O} \\
& \text{INPUT} \\
& \text{OUTPUT}
\}
\end{align*}
\]

<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 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 programeilen,  ! [1] length in bytes of name {0:31}
      programe[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
    exclusionspec,     ! [50] %00 if SHARED,  correc-
                        ! %20 if EXCLUSIVE,  spons
                        ! %60 if PROTECTED.  to flag
    accessspec,        ! [51] %0000 if I-O,
                        ! %2000 if INPUT,   param of
                        ! %4000 if OUTPUT.  OPEN.
    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> } , ••• ]
```
COMMAND INTERPRETER/APPLICATION INTERFACE
PARAM Command

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

  \[2 + \text{<parameter name length>} + \text{<parameter value length>}\]

  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 \(\text{REPLY} = 70\). 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;
    BEGIN
        INT msg^code,
        numparams;
        STRING parameters [ 0:1023 ];
    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

   \[
   \text{OPEN } <\text{receive depth}> \geq 1
   \]

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

   \[
   \text{OPEN } <\text{flags}>,<l> = 1
   \]

   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 \text{REPLY } <\text{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 ! ;
        ! 0 ! BEGIN ! startup message.
        .
          process startup message.
        .
        END;
        ! -1 ! 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.
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:

```plaintext
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:

```plaintext
STRUCT displayAmsg;
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.
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):

```
(CI1)   (CI2)   (CI3)  ...  (CiN)  COMMAND INTERPRETERS

($CMON)   CI MONITOR PROCESS (SERVER)
```

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 \( \geq 1 \), 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 logonAmsg;
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 logonAreply;
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.
COMMAND INTERPRETER/APPLICATION INTERFACE
Application-Supplied $CMON

LOGOFF MESSAGE: This message is sent to the $CMON process when a
LOGOFF command is entered.

The form of the logoff message is:

```c
STRUCT logoffAmsg;
BEGIN
  INT msgcode,
  userid,        ! [0] -51
  cipri,         ! [1] userid of user logging off.
  ciinfile [ 0:11 ],
  cioutfile[ 0:11 ];
END;
```

The length of this message is 54 bytes.

The form of the reply to the logoff message is:

```c
STRUCT logoffAreply;
BEGIN
  INT replycode;    ! [0] 0 or 1.
  STRING
    replytext[ 0:131 ];
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:

```c
STRUCT processcreationAmsg;
BEGIN
  INT msgcode,
  userid,        ! [0] -52.
  cipri,         ! [1] user ID of user logging on.
  ciinfile [ 0:11 ],
  cioutfile[ 0:11 ],
  progrname[ 0:11 ],
  priority,
  processor;    ! [27] expanded program file name.
  ! [39] <priority> of RUN command if
  ! supplied; otherwise -1.
END;
```

The length of this message is 82 bytes.

11-32
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 processcreationreply;
BEGIN
  INT replycode,     ! [0] 0 = create the process.
    programe [ 0:11 ], ! [1] expanded name of program file
    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 processcreationreply;
BEGIN
  INT replycode;     ! [0] 1 = disallow process creation.
    STRING           ! [1] optional message to be printed.
      replytext [ 0:131 ];
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:

```
  (REQUESTOR)  REQUEST  (SERVER)
```

```
  
  REPLY
```

Typically, a server process can accept requests from multiple requestor processes:

```
  (R1)
  (R2)
  .
  .
  .
  (RN)
```

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
Introduction

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 database as indicated by the supplied key value.

- "next^record^request": get the next record from the database 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.
NonStop PROGRAMMING EXAMPLE

Introduction

• "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 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 requester, of the current request
- the data record
- the error return value, by requester, of the preceding nonretryable request

Any time a failure of the server primary occurs, the backup takes over from the latest checkpoint (which is for the latest nonretryable operation) and reexecutes the latest nonretryable operation. (Note that this generates the result value for the latest nonretryable operation. The backup now has the correct values for any nonretryable operation which had been performed by the primary.) If the failure occurs between the call to READUPDATE and the call to CHECKPOINT, the backup is reexecuting an operation already replied to. If the failure occurs between the call to CHECKPOINT and the call to REPLY, the backup completes the operation associated with the current request. In either case, the use of the "sync block" by the file system ensures that the request is not duplicated. Note also that the backup executes the call to REPLY at the end of the processing loop. This call is rejected, because there is no outstanding request on the backup side at this time (the rejection is ignored).

If the primary server process was in the midst of processing a request when a failure occurred, the backup server process receives the same request when it takes over. If the primary had not reached the checkpoint for the current operation, then the backup has the value of the preceding nonretryable sync ID. The backup sees the request as a
new request and processes it. If the primary was processing a nonretryable request and had reached the checkpoint for the current operation, then the current sync ID was checkpointed (and, in fact, the backup completed the operation on its takeover). The backup sees this request as a duplicate request and returns the saved error code for the completed operation.
REQUEST CHECKPOINT

READUPDATE: A REQUEST

IS THIS A DUPLICATE, NONRETRYABLE REQUEST?

Y N

INSERT OR DELETE REQUEST (NONRETRYABLE)

CHECKPOINT
- Sync ID & Data
- Current Req.
- Sync Block for Database File
- Last Reply to Preceding Non-Retryable Req.

QUERY OR NEXT-RECORD REQUEST (RETRYABLE)

REPLY TO REQUEST
- For all request types, an error code is returned.
- For query and next-record requests, a data record is returned.
- For query and next-record requests, a data record is returned.

INSERT (WRITE) OR DELETE (WRITEUPDATE) RECORD

READ REQUESTED RECORD

1. Any failure will result in a duplicate request
2. A failure in this area results in the backup reexecuting the preceding nonretryable request
3. A failure in this area results in the backup reexecuting the current nonretryable request

Figure 12-2. Request Checkpoint
NonStop PROGRAMMING EXAMPLE

Introduction

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.)
NonStop Programming Example

for GUARDIAN Programming Manual

P/N 82096 (TNS and TNS/II)

Server Program

NO LIST

This program is run by the following RUN command:

RUN serveobj / IN <data base file> , NAME $SERVE /

where

<data base file> is the file name of the disc file to be accessed.

$SERVE is the name used by requestors to access this process.

process NonStop state.

INT backup.cpu, backup.pid[0:3], backup.up := 0, to detect looping backup deletescreates.

stop"count := 0, protected.

backup"up := 0, true if backup is running.

.stack"base; beginning of data stack for checkpointing.

files

INT recv"fnum, $RECEIVE file number.

.db"fnum, data base file number.

db"fname[0:11], data base file name.

LITERAL open"msgs = 40000,

protected = $40,

no"wait = 1,

recv"flags = open"msgs + no"wait,

recv"sync"depth = 1,

db"flags = protected,

db"sync"depth = 1;

.data base format.

LITERAL db"rec"len = 256,

db"rec"key"off = 0,

db"rec"key"len = 24;

.request message format.

word

[0] [1] [2:message size-3]

[ request type ] [ ------------------------- record -------------------- ]
57. sync = sync id (incremented by requester on each request).
58. used to detect duplicate nonretryable requests.
59. request type.
60. 0 = insert request (nonretryable)
61. 1 = delete request (nonretryable)
62. 2 = query request (retryable)
63. 3 = next record request (retryable)
64. record.
65. on request = record to be inserted, deleted, or obtained.
66. on return for query or next = record from database.
67. sync = sync id (incremented by requester on each request).
68. used to detect duplicate nonretryable requests.
69. request type.
70. 0 = insert request (nonretryable)
71. 1 = delete request (nonretryable)
72. 2 = query request (retryable)
73. 3 = next record request (retryable)
74. record.
75. on request = record to be inserted, deleted, or obtained.
76. on return for query or next = record from database.
77. sync = sync id (incremented by requester on each request).
78. used to detect duplicate nonretryable requests.
79. request type.
80. 0 = insert request (nonretryable)
81. 1 = delete request (nonretryable)
82. 2 = query request (retryable)
83. 3 = next record request (retryable)
84. record.
85. on request = record to be inserted, deleted, or obtained.
86. on return for query or next = record from database.
87. sync = sync id (incremented by requester on each request).
88. used to detect duplicate nonretryable requests.
89. request type.
90. 0 = insert request (nonretryable)
91. 1 = delete request (nonretryable)
92. 2 = query request (retryable)
93. 3 = next record request (retryable)
94. record.
95. on request = record to be inserted, deleted, or obtained.
96. on return for query or next = record from database.
97. sync = sync id (incremented by requester on each request).
98. used to detect duplicate nonretryable requests.
99. request type.
100. 0 = insert request (nonretryable)
101. 1 = delete request (nonretryable)
102. 2 = query request (retryable)
103. 3 = next record request (retryable)
104. record.
105. on request = record to be inserted, deleted, or obtained.
106. on return for query or next = record from database.
107. sync = sync id (incremented by requester on each request).
108. used to detect duplicate nonretryable requests.
109. request type.
110. 0 = insert request (nonretryable)
111. 1 = delete request (nonretryable)
112. 2 = query request (retryable)
113. 3 = next record request (retryable)
114. record.
115. on request = record to be inserted, deleted, or obtained.
116. on return for query or next = record from database.
117. sync = sync id (incremented by requester on each request).
118. used to detect duplicate nonretryable requests.
119. request type.
120. 0 = insert request (nonretryable)
121. 1 = delete request (nonretryable)
122. 2 = query request (retryable)
123. 3 = next record request (retryable)
124. record.
125. on request = record to be inserted, deleted, or obtained.
126. on return for query or next = record from database.
127. sync = sync id (incremented by requester on each request).
128. used to detect duplicate nonretryable requests.
129. request type.
130. 0 = insert request (nonretryable)
131. 1 = delete request (nonretryable)
132. 2 = query request (retryable)
133. 3 = next record request (retryable)
134. record.
135. on request = record to be inserted, deleted, or obtained.
136. on return for query or next = record from database.
137. sync = sync id (incremented by requester on each request).
138. used to detect duplicate nonretryable requests.
000767 0 0 ! entry_no[4] = <cpu, pin> of backup process, if any, or zero
000767 0 0 ?NOLIST
this procedure opens the $RECEIVE file for the primary process and reads the start-up message. Note that the receipt of the start-up message involves reading three interprocess messages:

1. OPEN "system" message.
2. CI start-up message (not a "system" message)
3. CLOSE "system" message.

PROC read"start"up"message;
BEGIN
INT .recv"buf[0:33];

open $RECEIVE.
recv"buf ':=' "$RECEIVE", El
CALL OPEN ( recv"buf, recv"fnum, recv"flags, recv"sync`depth );
IF <> THEN CALL ABEND;

read open message.
CALL READ ( recv"fnum, recv'buf, 14 ) ;
CALL AWAITIO ( recv"fnum,,,, 3000D); IF <> OR recv"buf <> -30 THEN CALL ABEND;

read start"up message.
CALL READ ( recv"fnum, recv~buf, 66 ) ;
CALL AWAITIO ( recv"fnum,,,, 3000D); IF <> OR recv"buf <> -1 THEN CALL ABEND;

save data"base file name.
db"fname ':=' recv"buf[9] FOR 12;

read close message.
CALL READ ( recv"fnum, recv"buf, 14 ) ;
CALL AWAITIO ( recv"fnum,,,, 3000D); IF <> OR recv"buf <> -31 THEN CALL ABEND;
END;

read"start"up"message.
! this procedure opens the data base file for the primary process.

000000 0 0 PROC open"primarys"files;
218. 000000 1 0 BEGIN
219. 000000 1 1 CALL OPEN ( db"fname, db"fnum, db"flags, db"sync"depth );
220. 000011 1 1 IF <> THEN CALL ABEND;
221. 000015 1 1 END; ! open"primarys"files.

000000 070012 070011 100040 100001 024733 002004 100360 024700 00010 027000 012003 000002 024711 027000 125003

this procedure opens the $RECEIVE and data base files for the backup

PROC open"backups"files;
228. 000000 1 0 BEGIN
230. 000000 1 1 INT .buf[0:11],
231. 000000 1 1 back"error;
232. 000000 1 1 BEGIN
233. 000000 1 1 ICON .buf[0:11],
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 IF <> THEN CALL STOP ( backup"pid );
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

DUP Variable INT L+002 Direct
   Variable INT L+001 Indirect

000000 070403 024700 002015 170401 000025 020043 002000 100014 00010 026047 170401 040010 005100 004001 100001 000002 070
000020 024766 100171 024700 027000 012004 070001 100001 024711 00030 027000 070012 040011 100040 100001 000002 070402 024
000040 100171 024700 027000 012004 070001 100001 024711 027000 00050 125003 000045 022122 042503 042511 053105 020040 020
000060 020040 020040 020040 020040 020040 020040 020004 020040 020040
This procedure creates the backup process. The steps involved are:

- Create backup.
- Open its files.
- Checkpoint current state.

```
PROC create_backup ( backup'cpu );

INT backup'cpu;
BEGIN
  INT .pfile[0:11],
      pname[0:3],
      error,
      status,
      .globals := 5; base for initial checkpoint = 'G'[5].
      "backup'cpu" and "backup'pid" not checkpointed.

  IF stop'count > 5 THEN CALL DEBUG;

  CALL PROGRAMFILENAME ( pfile );
  CALL GETCRTPID ( MYPID, pname );
  CALL NEWPROCESS ( pfile, "backup'cpu", backup'pid, error, pname );
  IF backup'pid THEN ! it was created.
     backup'up := 1;
     CALL open"backups"files;
     IF ( status := CHECKPOINT ( .globals, 'db'fnnum ) ) THEN *** CHECKPOINT *** !
     CALL analyze"checkpoint"status ( status );
END;
END;
```

**Variables:***
- BACKUP'CPU
- ERROR
- GLOBALS
- PFILE
- PNAME
- STATUS

```
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
```
PROC analyze_checkpoint_status (status);

INT status;

BEGIN
  IF backup_up THEN  
    analyze_it.
  ELSE
    CASE status.<0:7> OF
      BEGIN
        0 ! : ! good checkpoint.
        1 ! BEGIN ! checkpoint failure.
      END;
      2 ! BEGIN ! takeover from primary.
      3 ! BEGIN ! bad parameter to CHECKPOINT
      END; ! case of status.<0:7>.
      8-0 BEGIN ! primary stopped, so stop myself.
      8-1 BEGIN ! primary abended, so create a backup for me.
      8-2 BEGIN ! primary cpu down, note it.
      8-3 ! primary called CHECKSWITCH.
      8-4 ! case of status.<8:15>.
      END; ! 1.
    END;
    CASE status.<8:15> OF
      BEGIN
        2-0 BEGIN ! primary stopped, so stop myself.
        2-1 BEGIN ! primary abended, so create a backup for me.
        2-2 BEGIN ! primary cpu down, note it.
        2-3 ! primary called CHECKSWITCH.
        2-4 BEGIN ! case of status.<8:15>.
        END; ! 2.
      END;
      10-0 BEGIN ! bad parameter to CHECKPOINT
      END; ! 3.
    END;
    CALL DEBUG;
    END; ! 3.
  END; ! case of status.<0:7>.
END; ! analyze_checkpoint_status.
This procedure searches the requestor process directory by a process id

begin

proc lookuppid(pid);

begin

  int entryno := 0;

  complen := if pid.<0:7> = "$" then process name 3 else 4;

  while (entryno := entryno + 1) <= maxreqstrs do

    if pid = pids[entryno * 5] for complen then found it.

    return entryno;

  end;

  return 0; not found.

end; lookuppid.

variable

int

l+002

direct

p001

variable

int

l+001

direct

p003

variable

int

l+003

indirect
This procedure adds a process id to the requester process directory.

return values.

>0 = directory full, "pid" not added to directory.

0 = directory full, "pid" added, entry no of "pid" in directory.

INT PROC addpid(pid);

INT .pid;

BEGIN

INT entryNo, entryNo in local pid directory.

zero[0:3] := [ 4 * [ 0 ] ]; ! for lookup of empty directory slot.

IF (entryNo := lookuppid(pid)) THEN ! already in directory.

BEGIN

check for duplicate open.


BEGIN

END

ELSE ! not in directory. First open by "pid"

BEGIN

IF (entryNo := lookuppid(zero)) THEN ! look for empty slot.

BEGIN

pids[ entryNo * 5 ] := pid FOR 4;

syncCount[entryNo] := -1; ! initialize requestor idCount.

END;

RETURN entryNo; ! this returns zero if no room in directory.

END; ! addpid.

ENTRY^NO Variable INT L=001 Direct
PID Variable INT L=003 Indirect
ZERO Variable INT L=002 Direct
This procedure deletes a process id from the requestor process directory.

PROC delpid(pid);

INT .pid; /* pid to be deleted. */

BEGIN

INT entryAno; /* entryAno in local pid directory. */

IF (entryAno := lookuppid(pid)) THEN /* delete it. */

IF pids[entryAno * 5 + 4] THEN /* was open by process-pair. */

BEGIN

IF pids[entryAno * 5 + 3] = pid[3] THEN /* close by primary. */

BEGIN

END; /* delpid. */

ELSE /* was open by one process. */

BEGIN

pids[entryAno * 5] := [0, 0, 0, 0];

END; /* delpid. */

END; /* delpid. */
this procedure is called if a cpu failure message is received. It
deletes all references in the requestor process directory to the
failed cpu. This may cause entire entries to be deleted.

PROC delallpids(cpu);
INT cpu; // processor module number of pids to be deleted.
BEGIN
    INT entry"no := 0, // entry"no in local pid directory.
        temp;
    WHILE (entry"no := entry"no + 1) <= max"reqstrs DO
        BEGIN // check each entry"no.
            WHILE (entry"no := entry"no + 1) <= max"reqstrs DO
                BEGIN // check for match with entry"no's primary cpu.
                    IF pids[ entry"no * 5 + 3 ] AND pids[ entry"no * 5 + 3 ] < cpu THEN // primary down
                        CALL delpid ( pids [ entry"no * 5 ] )
                    END;
                END;
            END; // check for match with entry"no's backup cpu.
                CALL delpid ( pids [ entry"no * 5 ] )
            END;
        END;
    END;
END; // delallpids.
This procedure analyzes and takes appropriate action for system messages.

```plaintext
PROC analyze_system_message ( recvbuf, recvcnt, pid );

INT .recvbuf, .recvcnt, .pid;

BEGIN
    INT reply_error := 0, status;

    CASE $ABS ( recvbuf ) OF
        BEGIN ! cpu down.
            ! delete any references in the requestor process directory to the cpu that failed.
            CALL deleallpids ( recvbuf[l] );

            IF recvbuf[1] = backupcpu THEN backupup := 0;
            END;

        BEGIN ! cpu up.
            IF recvbuf[1] = backupcpu THEN backup cpu came up.
            BEGIN
                ! clear stop count.
                stopcount := 0;
            END;

            CALL create_backup ( backupcpu );

        BEGIN ! backup stopped.
            backupup := 0;
            stopcount := stopcount + 1;

            CALL create_backup ( backupcpu );

        BEGIN ! backup abended.
            backupup := 0;
            stopcount := stopcount + 1;

            CALL create_backup ( backupcpu );

        BEGIN ! OPEN system message.
            ! check for no-wait i/o depth > 1.
            IF recvbuf[1] .<12:15> > 1 THEN
                reply_error := 28 ! return illegal no-wait depth error.
                EXIT ELSE
                    try to add opener to directory.
                    IF NOT addpid ( pid ) THEN
                        reply_error := 12; ! return file in use error.
                        END;
```

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 OTHERWISE reply "error" = 2; ! return invalid operation error.
503.  000103 1 2 END; ! system message case.
504.  000154 1 1
505.  000154 1 1 IF ( status := CHECKPOINT ; stack base, 
506.  000154 1 1 max regs * 5 + 5, 
507.  000154 1 1 sync count [ 1 ], max regs, 
508.  000154 1 1 stop count , 1, 
509.  000154 1 1 reply "error" [ old requestor ], 1, 
510.  000154 1 1 db fnum ) ) THEN
511.  000154 1 1 CALL analyze "checkpoint" status ( status );
512.  000204 1 1 CALL analyze "system" message.
513.  000207 1 1 END; ! analyze "system" message.
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 00215 015
00020 100000 044006 010531 103001 143705 040000 000215 015005 00030 100000 044005 040000 024700 027000 010476 103001 143
00040 100001 074005 040000 024700 027000 010506 100000 044006 00500 100001 074005 040000 024700 027000 010476 103001 143
00060 006017 001001 016003 100034 044001 010406 170703 024700 00070 027000 015002 100014 044001 010457 170703 024700 027
00100 010453 100002 044001 010450 100037 000205 01102 000100 00110 100401 100040 000030 000041 000040 177672 177705 000
00120 177716 177725 000032 000031 000030 000227 000028 000025 00130 000024 000023 000022 000021 000020 000017 000016 000
00140 000014 000013 000012 000011 000010 000007 000006 000005 00150 000004 000003 000002 000001 000000 0000215 000015 000
00160 102001 172032 100200 070005 024755 100001 03101 071401 00170 100001 100000 040011 024744 020200 000007 004375 100
00200 024711 027000 034402 014403 040402 024700 027000 002004 00210 040401 100001 024711 027000 125006

NonStop PROGRAMMING EXAMPLE
Example Program Coding
This procedure searches the data base for the record associated with a key value.

Return values:

0 = record found, record in "result".

>0 = file management error.

```fortran
INT PROC queryArequest ( rec, result, resultrlen ) ;
INT .rec, .result, .resultrlen;
INT error;
resultrlen := 0;
CALL KEYPOSITION ( dbAfnum, rec,,,2 );
CALL READUPDATE ( dbAfnum, result, dbArecrlen, resultrlen );
CALL FILEINFO ( dbAfnum, error ) ;
RETURN error;
END;
```

error: queryArequest.

Variable INT L=001 Direct
REC Variable INT L=005 Indirect
RESULT Variable INT L=004 Indirect
RESULTrlen Variable INT L=003 Indirect

00000 002001 100000 144703 040011 170705 030001 000002 100002 00010 100031 024755 027000 040011 170704 005001 170703 000020 024755 100036 024700 027000 040011 070401 024711 002013 00030 005030 024700 027000 040401 125006
541. 00000 0 0 | this procedure adds a record to the data base.
542. 00000 0 0 |
543. 00000 0 0 | return values.
544. 00000 0 0 |
545. 00000 0 0 | 0 = record "rec" added.
546. 00000 0 0 | >0 = file management error.
547. 00000 0 0 |
548. 00000 0 0 INT PROC insert"request ( rec ) :
549. 00000 1 0 INT .rec; | record to be added.
550. 00000 1 0 |
551. 00000 1 1 BEGIN
552. 00000 1 1 |
553. 00000 1 1 INT error;
554. 00000 1 1 |
555. 00000 1 1 | insert the record.
556. 00000 1 1 CALL WRITE ( db"fnum, rec, db"rec"len );
557. 00001 1 1 CALL FILEINFO ( db"fnum, error );
558. 00020 1 1 RETURN error;
559. 00022 1 1 END; | insert"request.

ERROR
REC Variable INT L+001 Direct
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 database 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 deleteArequest ( 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;  ! deleteArequest.
Procedure next_record_request

```
INT PROC next_record_request ( rec, result, result_len );
INT rec,   ! key of record for positioning.
INT .result,   ! next record.
INT .result_len;   ! length of next record.
BEGIN
INT error;
STRING .srec := @rec '<<' 1;
.srec [ db->rec->key->off + db->rec->key->len - 1 ] :=
.srec [ db->rec->key->off + db->rec->key->len - 1 ] '+' 1;
CALL KEYPOSITION ( db->fnum, rec );
CALL READ ( db->fnum, result, db->rec->len, result_len );
CALL FILEINFO ( db->fnum, error );
RETURN error;
END
```

**Variables**
- REC INT 1-005 Indirect
- RESULT INT 1-003 Indirect
- RESULT_LEN INT 1-003 Indirect
- SREC STRING 1-002 Indirect
- ERROR Variable INT L+001 Direct
PROC process_user_request { recvAbuf, recvAcnt, pid }

INT .recvAbuf, .recvAcnt, .pid; // process id of requester.

BEGIN

INT status, requester, replyAlen := 0; // reply length for current request.

BEGIN

IF NOT { requester := lookuppid { pid } ) THEN

CALL REPLY (,,, 60 ); // return "device has been downed" error.

RETURN;

END;

! check for duplicate request.

IF recvAbuf [ requestAtype ] <= 1 THEN ! non-retryable,

BEGIN

! save sync count of current requester.

syncAcount [ requester ] := recvAbuf;

IF ( status := CHECKPOINT ( stackAbase, dbAfnum, syncAcount [ requester ], 1, replyAerror [ oldArequestor ], 1, recvAbuf, ( recvAcnt + 1 ) / 2 ) ) THEN

CALL analyzeAcheckpointAstatus ( status ) ;

! save requester number of current requester for a subsequent checkpoint.

oldArequestor := requester;

END; // of checkpoint.

END; // of request

BEGIN

! save request type.

replyAbuf ':=' recvAbuf FOR 2;

BEGIN

CALL processAbaseArequest ( recvAbuf [ requestAtype ] ) OF

BEGIN

! "insert" request.

replyAerror [ requester ] := insertArequest ( recvAbuf [ record ] );

! "delete" request.

replyAerror [ requester ] := deleteArequest ( recvAbuf [ record ] );

! "query" request.

replyAerror [ requester ] := queryArequest ( recvAbuf [ record ], replyAbuf [ record ], replyAlen );

! "next entry" request.

END.
672. 000123 1 3
673. 000123 1 3
674. 000134 1 3
675. 000134 1 3
676. 000137 1 3
677. 000154 1 2
678. 000154 1 1
679. 000154 1 1
680. 000154 1 1
681. 000165 1 1
682. 000165 1 1

<table>
<thead>
<tr>
<th>PID</th>
<th>Variable</th>
<th>INT</th>
<th>L-003</th>
<th>Indirect</th>
</tr>
</thead>
<tbody>
<tr>
<td>RECV&quot;BUF</td>
<td>Variable</td>
<td>INT</td>
<td>L-005</td>
<td>Indirect</td>
</tr>
<tr>
<td>REPLY&quot;LEN</td>
<td>Variable</td>
<td>INT</td>
<td>L-003</td>
<td>Direct</td>
</tr>
<tr>
<td>REQUESTOR</td>
<td>Variable</td>
<td>INT</td>
<td>L+002</td>
<td>Direct</td>
</tr>
<tr>
<td>STATUS</td>
<td>Variable</td>
<td>INT</td>
<td>L+001</td>
<td>Direct</td>
</tr>
</tbody>
</table>

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 01103E 140705 00030 147032 170007 100000 040011 173032 100001 024744 031
00040 171033 100001 170705 040704 104001 100002 002113 024733 00050 002022 005005 004374 100000 024711 027000 034401 014
00060 040401 024700 027000 040402 044031 170304 170705 100002 00070 026007 103001 143705 010444 103002 173705 024700 027
00100 033402 147033 010451 103002 173705 024700 027000 033402 01110 147033 010442 103002 173705 173034 070403 024722 0027
00120 033402 147033 010431 103002 173705 173034 070402 024702 01130 027000 033402 147033 010420 033402 100035 147033 010
00140 100003 002005 011002 000100 010401 100004 00003C 177725 00150 177733 177741 177751 177761 170034 040403 104004 000
00160 033402 143033 100031 024755 027000 125006
1--1

PAGE 21 $BOOKSL.M096B01.SERVESRC [l] SERVE; execute Procedure 2/23/82 13:

684. 000000 0 0 ! this is the main execution loop of the server process. The server waits
685. 000000 0 0 ! for incoming requests or system messages.
686. 000000 0 0 PROC execute;
687. 000000 1 0 BEGIN
688. 000000 1 1 INT .pid[0:3], ! requestor <process id>.
689. 000000 1 1 INT system.message, ! system message.
690. 000000 1 1 INT status;
691. 000000 1 1 WHILE 1 DO ! loop on requests.
692. 000000 1 2 BEGIN
693. 000000 1 3 WHILE 1 DO ! loop on requests.
694. 000000 1 4 WHILE 1 DO ! loop on requests.
695. 000000 1 5 ! read $RECEIVE file.
696. 000000 1 6 CALL READUPDATE ( recv"fnum, recv"buf, message"len );
697. 000000 1 7 CALL AWAITIO ( recv"fnum, recv"cnt );
698. 000000 1 8 IF >= THEN ! read a message.
699. 000000 1 9 BEGIN
700. 000000 2 0 system.message := >; ! save system message condition.
701. 000000 2 1 CALL LASTRECEIVE ( pid );
702. 000000 2 2 CALL analyze"system"message ( recv"buf, recv"cnt, pid )
703. 000000 2 3 IF system.message THEN
704. 000000 2 4 CALL process"user"request ( recv"buf, recv"cnt, pid )
705. 000000 2 5 ELSE
706. 000000 2 6 CALL process"user"request ( recv"buf, recv"cnt, pid )
707. 000000 2 7 END; ! loop on requests.
708. 000000 2 8 END; ! loop on requests.
709. 000000 2 9 END; ! loop on requests.
710. 000000 3 0 END; ! execute.

PID STATUS SYSTEM.MESSAGE

00000 070404 024700 002006 040010 070026 050001 040004 024722
00010 002003 100034 024700 027000 070010 100000 070027 024
00020 002003 100024 024700 027000 010427 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

NonStop PROGRAMMING EXAMPLE Program Coding
I-1. 00000 0 0 0; this is the "main". This is where the primary/backup determination is
I-2. 00000 0 0 0 made.
I-3. 00000 0 0 0 PROC serve MAIN;
I-4. 00000 1 0 BEGIN
I-5. 00000 1 1 INT base = 'L' + 1,
I-6. 00000 1 1 .ppdentry[0:8];
I-7. 00000 1 1 @stack"base := @base;
I-8. 00000 1 1 CALL ARMTRAP ( 0, -1 );
I-9. 00000 1 1 ! get process name.
I-10. 00000 1 1 CALL GETCRTPID ( MYPID, ppdentry );
I-11. 00000 1 1 CALL LOOKUPPROCESSNAME ( ppdentry );
I-12. 00000 1 1 IF < THEN CALL ABEND; ! not named.
I-13. 00000 1 1 ! calculate backup cpu number. cpu's are paired 0-1, 2-3, 4-5, ...
I-14. 00000 1 1 backup"cpu := MYPID.<0:7>;
I-15. 00000 1 1 backup"cpu.<15> := NOT backup"cpu.<15>;
I-16. 00000 1 1 ! monitor all cpus.
I-17. 00000 1 1 CALL MONITORCPUS ( -1 );
I-18. 00000 1 1
I-20. 00000 1 1 BEGIN
I-21. 00000 1 2 CALL read"startup"message;
I-22. 00000 1 2 CALL open"primarys"files;
I-23. 00000 1 2 CALL create"backup ( backup"cpu );
I-24. 00000 1 2 CALL execute;
I-25. 00000 1 2 END;
I-26. 00000 1 1 ELSE ! im the backup.
I-27. 00000 1 1 BEGIN
I-28. 00000 1 2 ! wait for failure
I-29. 00000 1 2 CALL CHECKMONITOR;
I-30. 00000 1 2 CALL ABEND;
I-31. 00000 1 2 END;
I-32. 00000 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 100010 070000 000
00040  100777 024700 027000 103004 141401 015407 027000 027000 00050 040000 024700 027000 027000 027000 010405 027000 000107 000
00060  024711 027000 000002 024711 127000

NonStop PROGRAMMING EXAMPLE
Example Program Coding
ABEND Proc
ADDPID Proc INT
ANALY"CHECKPOINT"STATUS Proc
ANALY"SYSTEM"MESSAGE Proc
ARMTRAP Proc
BACK"CPU Variable INT G+000 Direct
BACK"PID Variable INT G+001 Direct
BACK"UP Variable INT G+006 Direct
BUP Proc INT
CHECKMONITOR Proc
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
DELETE"REQUEST Proc INT
DELPID Proc
EXECUTE Proc
FILEINFO Proc
GETCRTPID Proc
INSERT"REQUEST Proc INT
KEYPOSITION Proc
LASTRECEIVE Proc
LOOKUP"PID Proc
LOOKUP"PROCESSNAME Proc
MAX"REQUESTS Literal %000020
MESSAGE"LEN Literal %000040
MESSAGE"SIZE Literal %000202
MOM Proc
MONITORCPUS Proc
MYPID Proc INT
NEWPROCESS Proc
NEXT"RECORD"REQUEST Proc INT
NEXT"REQUESTOR"REQUEST Proc
NO"WAIT Literal INT G+00001 Direct
OLD"REQUESTOR Variable INT G+031 Direct
OPEN Proc
OPEN"BACKUPS"FILES Proc
OPEN"MSGS Proc
OPEN"PRIMARYS"FILES Proc
PIDS Variable INT G+035 Indirect
PROCESS"USER"REQUEST Proc
PROGRAMFILENAME Proc
PROTECTED Literal %000040
QUERY"REQUEST Proc
READ Proc
READUPDATE Proc
READ"START"UP"MESSAGE Proc
RECORD Literal %000002
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Type</th>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RECV'BUF</td>
<td>Variable</td>
<td>G+026</td>
<td>Indirect</td>
</tr>
<tr>
<td>RECV'CNT</td>
<td>Variable</td>
<td>G+027</td>
<td>Direct</td>
</tr>
<tr>
<td>RECV'FLAGS</td>
<td>Literal</td>
<td>G+040001</td>
<td></td>
</tr>
<tr>
<td>RECV'FNUM</td>
<td>Variable</td>
<td>G+010</td>
<td>Direct</td>
</tr>
<tr>
<td>RECV'SYNC'DEPTH</td>
<td>Literal</td>
<td>G+000001</td>
<td></td>
</tr>
<tr>
<td>REPLY</td>
<td>Proc</td>
<td></td>
<td></td>
</tr>
<tr>
<td>REPLY'BUF</td>
<td>Variable</td>
<td>G+034</td>
<td>Indirect</td>
</tr>
<tr>
<td>REPLY'ERROR</td>
<td>Variable</td>
<td>G+033</td>
<td>Indirect</td>
</tr>
<tr>
<td>REQUEST'TYPE</td>
<td>Literal</td>
<td>G+000001</td>
<td></td>
</tr>
<tr>
<td>SERVE</td>
<td>Proc</td>
<td></td>
<td></td>
</tr>
<tr>
<td>STACK'BASE</td>
<td>Variable</td>
<td>G+007</td>
<td>Indirect</td>
</tr>
<tr>
<td>STOP</td>
<td>Proc</td>
<td></td>
<td></td>
</tr>
<tr>
<td>STOP'COUNT</td>
<td>Variable</td>
<td>G+005</td>
<td>Direct</td>
</tr>
<tr>
<td>SYNC</td>
<td>Literal</td>
<td>G+000000</td>
<td></td>
</tr>
<tr>
<td>SYNC'COUNT</td>
<td>Variable</td>
<td>G+032</td>
<td>Indirect</td>
</tr>
<tr>
<td>WRITE</td>
<td>Proc</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WRITEUPDATE</td>
<td>Proc</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PEP</td>
<td>BASE</td>
<td>LIMIT</td>
<td>ENTRY</td>
</tr>
<tr>
<td>-----</td>
<td>--------</td>
<td>-------</td>
<td>-------</td>
</tr>
<tr>
<td>002</td>
<td>000512</td>
<td>000600</td>
<td>000516</td>
</tr>
<tr>
<td>003</td>
<td>000370</td>
<td>000453</td>
<td>000370</td>
</tr>
<tr>
<td>004</td>
<td>000733</td>
<td>001147</td>
<td>000733</td>
</tr>
<tr>
<td>005</td>
<td>000311</td>
<td>000367</td>
<td>000311</td>
</tr>
<tr>
<td>006</td>
<td>000662</td>
<td>000732</td>
<td>000662</td>
</tr>
<tr>
<td>007</td>
<td>001227</td>
<td>001260</td>
<td>001227</td>
</tr>
<tr>
<td>010</td>
<td>000601</td>
<td>000661</td>
<td>000601</td>
</tr>
<tr>
<td>011</td>
<td>001511</td>
<td>001566</td>
<td>001511</td>
</tr>
<tr>
<td>012</td>
<td>001205</td>
<td>001226</td>
<td>001205</td>
</tr>
<tr>
<td>013</td>
<td>000454</td>
<td>000511</td>
<td>000454</td>
</tr>
<tr>
<td>014</td>
<td>001261</td>
<td>001322</td>
<td>001261</td>
</tr>
<tr>
<td>015</td>
<td>000223</td>
<td>000310</td>
<td>000223</td>
</tr>
<tr>
<td>016</td>
<td>000205</td>
<td>000222</td>
<td>000205</td>
</tr>
<tr>
<td>017</td>
<td>001323</td>
<td>001510</td>
<td>001323</td>
</tr>
<tr>
<td>020</td>
<td>001150</td>
<td>001204</td>
<td>001150</td>
</tr>
<tr>
<td>021</td>
<td>000023</td>
<td>000204</td>
<td>000023</td>
</tr>
<tr>
<td>022</td>
<td>001567</td>
<td>001653</td>
<td>001567</td>
</tr>
<tr>
<td>PEP</td>
<td>BASE</td>
<td>LIMIT</td>
<td>ENTRY</td>
</tr>
<tr>
<td>-----</td>
<td>------</td>
<td>-------</td>
<td>-------</td>
</tr>
<tr>
<td>021</td>
<td>000023</td>
<td>000024</td>
<td>000023</td>
</tr>
<tr>
<td>016</td>
<td>000205</td>
<td>000222</td>
<td>000205</td>
</tr>
<tr>
<td>015</td>
<td>000223</td>
<td>000310</td>
<td>000223</td>
</tr>
<tr>
<td>005</td>
<td>000311</td>
<td>000367</td>
<td>000311</td>
</tr>
<tr>
<td>003</td>
<td>000370</td>
<td>000453</td>
<td>000370</td>
</tr>
<tr>
<td>013</td>
<td>000454</td>
<td>000511</td>
<td>000454</td>
</tr>
<tr>
<td>002</td>
<td>000512</td>
<td>000600</td>
<td>000516</td>
</tr>
<tr>
<td>010</td>
<td>000601</td>
<td>000661</td>
<td>000601</td>
</tr>
<tr>
<td>006</td>
<td>000662</td>
<td>000732</td>
<td>000662</td>
</tr>
<tr>
<td>004</td>
<td>000733</td>
<td>001147</td>
<td>000733</td>
</tr>
<tr>
<td>020</td>
<td>001150</td>
<td>001204</td>
<td>001150</td>
</tr>
<tr>
<td>012</td>
<td>001205</td>
<td>001226</td>
<td>001205</td>
</tr>
<tr>
<td>007</td>
<td>001227</td>
<td>001260</td>
<td>001227</td>
</tr>
<tr>
<td>014</td>
<td>001261</td>
<td>001322</td>
<td>001261</td>
</tr>
<tr>
<td>017</td>
<td>001323</td>
<td>001510</td>
<td>001323</td>
</tr>
<tr>
<td>011</td>
<td>001511</td>
<td>001566</td>
<td>001511</td>
</tr>
<tr>
<td>022</td>
<td>001567</td>
<td>001653</td>
<td>001567</td>
</tr>
</tbody>
</table>
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
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**


**CALL ACTIVATEPROCESS** ( <process id> )


<status> := ALLOCATESEGMENT ( <segment id>, <segment size>, <file name>, <pin> ) (II only)

CALL ALTERPRIORITY ( <process id> , <priority> )


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> )


! INT: function ! BLINK^SCREEN ( @<screen name> , <buffer> , <field name> , <blink> )


CALL CANCEL ( <file number> )


CALL CANCELREQ ( <file number> , <tag> )


CALL CANCELTIMEOUT ( <tleaddress> )  (II only)

CALL CHANGELIST ( <file number>, <function>, <parameter> )


<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>
  I <file number>
  I <flags>
  I <sync or receive depth>
  I <sequential block buffer>
  I <buffer length>
  I <back error> )

See "Checkpointing Facility" section of the GUARDIAN Operating System Programming Manual.
APPENDIX A: PROCEDURE SYNTAX SUMMARY

```
{ <status> := } CHECKPOINT ( <stack base>
    , <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>
    , <descriptors> )

See "Checkpointing Facility" section of the GUARDIAN Operating System Programming Manual.

! INT: function ! CHECK^SCREEN ( @<screen name>
    , SCREEN
    , <buffer>
    , <check procedure> )


{ <status> := } CHECKSWITCH

See "Checkpointing Facility" section of the GUARDIAN Operating System Programming Manual.

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


{ CALL
    <error> := } CLOSE^FILE ( { <common fcb> }
    { <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> )


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


CALL CONTROLBUF ( <file number> , <operation>
   <buffer> , <count>
   <count transferred>
   <tag> )


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> )

CALL CREATEPROCESSNAME ( <process name> )


CALL CREATERNOTENAME ( <name> , <system number> )

See "Process Control" section of the GUARDIAN Operating System Programming Manual, or see the EXPAND Users Manual.

<accessor id> := CREATORACCESSID


CALL DEALLOCATESEGMENT ( <segment id> , <flags> ) (II only)


CALL DEBUG


CALL DEFINELIST ( <file number> , <address list>

, <address size> , <num entries>

, <polling count> , <polling type> )


<status> := DEFINEPOOL ( <pool head> , <pool> , <pool size> ) (II only)


CALL DELAY ( <time period> )

CALL DEVICEINFO ( <file name>, <device type>, <physical record length> )


<status> := EDITREAD ( <edit control block> , <buffer> , <buffer length> , <sequence number> )


<status> := EDITREADINIT ( <edit control block> , <file number> , <buffer length> )


CALL ENFORMFINISH ( <ctlblock> )

See the ENFORM Reference Manual.

{ <count> := } ENFORMRECEIVE ( <ctlblock> , <buffer> )

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.
APPENDIX A: PROCEDURE SYNTAX SUMMARY

! INT:function! EXPAND^SCREEN ( @<screen name> , SCREEN
    , <buffer>
    , <rewrite form> )


<status> := FILEERROR ( <file number> )


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> )


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-sequence params>
    , <alternate key params>
    , <partition params> )

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


! 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.
APPENDIX A: PROCEDURE SYNTAX SUMMARY

{ <error> := } FORMATDATA { <buffer> 
  | <bufferlen> 
  | <bufferoccurs> 
  | <length> 
  | <iformat> 
  | <variablelist> 
  | <variablelistlength> 
  | <flags> } 

See "Formatter" section of the GUARDIAN Operating System Programming Manual.

CALL GETCRTPID ( <cpu, pin>, <process id> ) 


<status> := GETDEVNAME ( <logical device no>, <device name> , <system number> )


<address> := GETPOOL ( <pool head>, <block size> ) (II only)


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 GETREMETECRTPID ( <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.

{l<1dev> := } GETSYSTEMNAME ( <system number>, <system name> ) { CALL

See "File System Procedures" section of the GUARDIAN Operating System Programming Manual, or see the EXPAND Users 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> )


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


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

CALL KEYPOSITION ( <file number>, <key value>, <key specifier>, <length word>, <positioning mode> )


<last address> := LASTADDR


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


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

See "File System Procedures" section of the GUARDIAN Operating System Programming Manual, or see the EXPAND Users Manual.

{<state> := } LOCKDATA ( <address>, <count>, <sys map> )

See "Advanced Memory Management" section of the GUARDIAN Operating System Programming Manual.

CALL LOCKFILE ( <file number>, <tag> )

APPENDIX A: PROCEDURE SYNTAX SUMMARY

<status> := LOCKMEMORY ( <address> , <byte count> , <timeoutvalue> , <parameter1> , <parameter2> )

See "Advanced Memory Management" section of the GUARDIAN Operating System Programming Manual.

CALL LOCKREC ( <file number> , <tag> )


CALL LOOKUPPROCESSNAME ( <ppd entry> )


CALL MOM ( <process id> )


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> )

<my cpu,pin> := MYPID


<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> )


CALL NEWPROCESSNOWAIT ( <filenames>, <priority>, <memory pages>, <processor>, <process id>, <error>, <name> ) (II only)


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

APPENDIX A: PROCEDURE SYNTAX SUMMARY

{ CALL } NOERROR ( <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> }


CALL NUMOUT ( <ascii result> \\
<unsigned integer> \\
<base> \\
<width> )


CALL OPEN ( <file name> \\
<file number> \\
<flags> \\
<sync or receive depth> \\
<primary file number> , <primary process id> \\
<sequential block buffer> , <buffer length> )


{ 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> )


! INT: function ! POSITION^SCREEN ( @<screen name> SCREEN , <buffer> , <field name> )


<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> SCREEN , <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> SCREEN , <data line> , <read count> , <count read> , <page number> )

See "Print Processes" section of the Spooler System Management Guide.
APPENDIX A: PROCEDURE SYNTAX SUMMARY

\[\text{<error code>} := \text{PRINTREADCOMMAND} (\text{<print control buffer>},\text{<control number>},\text{<device>},\text{<dev flags>},\text{<dev param>},\text{<dev width>},\text{<skip num>},\text{<data file>},\text{<job number>},\text{<location>},\text{<form name>},\text{<report name>},\text{<pagesize>})\]

See "Print Processes" section of the Spooler System Management Guide.

\[\text{<error code>} := \text{PRINTSTART} (\text{<job buffer>},\text{<print control buffer>},\text{<data file number>})\]

See "Print Processes" section of the Spooler System Management Guide.

\[\text{<error code>} := \text{PRINTSTATUS} (\text{<file number to Supervisor>},\text{<print control buffer>},\text{<message type>},\text{<device>},\text{<error>},\text{<num copies>},\text{<page>},\text{<line>},\text{<lines printed>})\]

See "Print Processes" section of the Spooler System Management Guide.

\[\text{<last priority>} := \text{PRIORITY} (\text{<priority>})\]


\[\text{<accessor id>} := \text{PROCESSACCESSID}\]

APPENDIX A: PROCEDURE SYNTAX SUMMARY

\[
\{ \text{<error>} := \} \text{PROCESSINFO} \{ \text{<cpu,pin>} \\
\{ \text{CALL} \\
, \text{<process id>} \\
, \text{<creator accessor id>} \\
, \text{<process accessor id>} \\
, \text{<priority>} \\
, \text{<program file name>} \\
, \text{<home terminal>} \\
, \text{<system number>} \\
, \text{<search mode>} \} \\
\]

See "Process Control" section of the GUARDIAN Operating System Programming Manual, or see the EXPAND Users Manual.

\(<\text{processor status}> := \text{PROCESSORSTATUS}\)

See "Checkpointing Facility" section of the GUARDIAN Operating System Programming Manual, or see the EXPAND Users Manual.

CALL PROGRAMFILENAME ( <program file> )


CALL PURGE ( <file name> )


CALL PUTPOOL ( <pool head> , <address> ) \hspace{1cm} \text{(II only)}


CALL READ ( <file number> \\
, <buffer> \\
, <read count> \\
, <count read> \\
, <tag> )

APPENDIX A: PROCEDURE SYNTAX SUMMARY

{ CALL { 
{ <error> := } READ^FILE ( <file fcb>, <buffer>, <count read>, 
<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> )


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


CALL READUPDATELOCK ( <file number>, <buffer>, <read count>, <count read>, <tag> )


CALL RECEIVEINFO ( <process id>, <message tag>, <sync id>, <file number>, <read count> )

APPENDIX A:  PROCEDURE SYNTAX SUMMARY

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

CALL


<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> )


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


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


CALL RESERVELCBS ( <no. receive lcbs> , <no. send lcbs> )


CALL RESETSYNC ( <file number> )

See "Checkpointing Facility" section of the GUARDIAN Operating System Programming Manual.
CALL SAVEPOSITION ( <file number>, <positioning block>, <positioning block size> )


\{
CALL \} SET^FILE ( \{ <common fcb> \}, <operation> \{ <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> )


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


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


CALL SETMYTERM ( <terminal name> )

CALL SETPARAM ( <file number>, <function>, <param array>, <count>, <last param array>, <last count> )


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

CALL


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

See "Checkpointing Facility" section of the GUARDIAN Operating System Programming Manual.

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


CALL SIGNALTIMEOUT ( <timeoutvalue>, <parameter1>, <parameter2>, <tleaddress> ) (II only)


{ <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.
APPENDIX A: PROCEDURE SYNTAX SUMMARY

\[
\begin{align*}
\text{SORTMERGEFINISH} & \{ \text{ctlblock} \}, \text{abort}, \text{spare 1}, \text{spare 2} \\
\text{SORTMERGERECIEVE} & \{ \text{ctlblock} \}, \text{buffer}, \text{length}, \text{spare 1}, \text{spare 2} \\
\text{SORTMERGESEND} & \{ \text{ctlblock} \}, \text{buffer}, \text{length}, \text{stream id}, \text{spare 1}, \text{spare 2} \\
\text{SORTMERGESTART} & \{ \text{ctlblock} \}, \text{key block}, \text{number merge files}, \text{number sort files}, \text{in file name}, \text{in file exclusion mode}, \text{in file count}, \text{in file length}, \text{format}, \text{out file name}, \text{out file exclusion mode}, \text{out file type}, \text{flags}, \text{errnum}, \text{errproc}, \text{scratch file name}, \text{scratch block}, \text{process start}, \text{max record length}, \text{collate sequence table}, \text{spare 1}, \text{spare 2}, \text{spare 3}, \text{spare 4}, \text{spare 5} \\
\end{align*}
\]

See "Programmatic Mode" section of the SORT/MERGE Users Guide.
APPENDIX A: PROCEDURE SYNTAX SUMMARY

\[
\{ \text{<error> := } \} \text{ SORTMGERGSTATISTICS ( <ctlblock>}
\]
\[
\text{CALL , <length>, <statistics>,}<\text{<spare 1>}
\]
\[
\text{<spare 2> })
\]

See "Programmatic Mode" section of the SORT/PERUSE Users Guide.

\[
\text{<error code> := SPOOLCONTROL ( <level 3 buffer>}
\]
\[
\text{, <operation>, <parameter>}
\]
\[
\text{, <bytes written to buffer> )}
\]

See "Spooler Interface Procedures" section of the Spooler/PERUSE Users Guide.

\[
\text{<error code> := SPOOLCONTROLBUF ( <level 3 buffer>}
\]
\[
\text{, <operation>, <buffer>}
\]
\[
\text{, <count>}
\]
\[
\text{, <bytes written to buffer> )}
\]

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> := SPOOLERCMD ( <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.
APPENDIX A:  PROCEDURE SYNTAX SUMMARY

<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> )


CALL STOP ( <process id> )


CALL SUSPENDPROCESS ( <process id> )


CALL TAKEBREAK ( <file fcb> )

See "Sequential I/O Procedures" section of the GUARDIAN Operating System Programming Manual.

CALL TIME ( <date and time> )


A-26
APPENDIX A: PROCEDURE SYNTAX SUMMARY

CALL TIMESTAMP ( <interval clock> )

<version> := TOSVERSION

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

CALL UNLOCKMEMORY ( <address> , <byte count> ) (II only)
See "Advanced Memory Management" section of the GUARDIAN Operating System Programming Manual.

CALL UNLOCKREC ( <file number> , <tag> )

CALL USERIDTOUSERNAME ( <id name> )

CALL USERNAMETOUSERID ( <name id> )
<old segment id> := USESEGMENT ( <segment id> , <pin> )  
                      (II only)


CALL VERIFYUSER ( <user name or id> , <logon> , <default> , <default length> )


<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> )


{ CALL := } WRITE^FILE ( <file fcb> , <buffer> , <write count> , <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> )


CALL WRITEUPDATE ( <file number>
, <buffer>
, <write count>
, <count written>
, <tag> )


CALL WRITEUPDATEUNLOCK ( <file number>
, <buffer>
, <write count>
, <count written>
, <tag> )

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)
APPENDIX B: FILE SYSTEM ERROR SUMMARY

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".
# APPENDIX B: FILE SYSTEM ERROR SUMMARY

## FILE SYSTEM ERROR SUMMARY

<table>
<thead>
<tr>
<th>Error Number</th>
<th>Description</th>
<th>Device Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CONDITION CODE = (CCE): NO ERROR</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>operation successful</td>
<td>any</td>
</tr>
<tr>
<td><strong>CONDITION CODE &gt; (CCG): WARNINGS</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>end-of-file or end of medium</td>
<td>3, 4, 6, 8</td>
</tr>
<tr>
<td>2</td>
<td>operation not allowed on this type file</td>
<td>any</td>
</tr>
<tr>
<td>3</td>
<td>failure to open or purge a partition</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>failure to open an alternate key file</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>failure to provide sequential buffering (I only)</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>system message received</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>process not accepting CONTROL, SETMODE, or RESETSYNC messages</td>
<td>0</td>
</tr>
<tr>
<td>8 (%10)</td>
<td>operation successful (examine MCW for additional status)</td>
<td>11.40, 11.42</td>
</tr>
<tr>
<td><strong>CONDITION CODE &lt; (CCL): ERRORS</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10 (%12)</td>
<td>file or record already exists</td>
<td>3</td>
</tr>
<tr>
<td>11 (%13)</td>
<td>file not in directory or record not in file</td>
<td>3</td>
</tr>
<tr>
<td>12 (%14)</td>
<td>file in use</td>
<td>any except 2</td>
</tr>
<tr>
<td>13 (%15)</td>
<td>illegal filename specification</td>
<td>any</td>
</tr>
<tr>
<td>14 (%16)</td>
<td>device does not exist</td>
<td>any except 1 and 2</td>
</tr>
<tr>
<td>15 (%17)</td>
<td>RENAME attempted to another volume</td>
<td>3</td>
</tr>
</tbody>
</table>
APPENDIX B: FILE SYSTEM ERROR SUMMARY

<table>
<thead>
<tr>
<th>Error Number</th>
<th>Description</th>
<th>Device Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 (%20)</td>
<td>file number has not been opened</td>
<td>any</td>
</tr>
<tr>
<td>17 (%21)</td>
<td>attempted CHECKOPEN when file not open by primary, CHECKOPEN parameters do</td>
<td>any</td>
</tr>
<tr>
<td></td>
<td>not match those of primary open, or primary process not alive</td>
<td></td>
</tr>
<tr>
<td>18 (%22)</td>
<td>referenced system does not exist</td>
<td>any</td>
</tr>
<tr>
<td>19 (%23)</td>
<td>no more devices in logical device table</td>
<td>any except</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 and 2</td>
</tr>
<tr>
<td>20 (%24)</td>
<td>attempted network access by process with five-character name or seven-</td>
<td>any except</td>
</tr>
<tr>
<td></td>
<td>character home terminal name</td>
<td>2</td>
</tr>
<tr>
<td>21 (%25)</td>
<td>illegal count specified</td>
<td>any except</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>22 (%26)</td>
<td>application parameter or buffer address out of bounds</td>
<td>any</td>
</tr>
<tr>
<td>23 (%27)</td>
<td>disc address out of bounds</td>
<td>3</td>
</tr>
<tr>
<td>24 (%30)</td>
<td>privileged mode required for this operation</td>
<td>any</td>
</tr>
<tr>
<td>25 (%31)</td>
<td>AWAITIO or CANCEL attempted on wait file</td>
<td>any</td>
</tr>
<tr>
<td>26 (%32)</td>
<td>AWAITIO, CANCEL, or CONTROL 22 attempted on a file with no outstanding</td>
<td>any</td>
</tr>
<tr>
<td></td>
<td>requests</td>
<td></td>
</tr>
<tr>
<td>27 (%33)</td>
<td>wait operation attempted when outstanding requests pending</td>
<td>any</td>
</tr>
<tr>
<td>28 (%34)</td>
<td>number of outstanding no-wait operations exceed OPEN specification, or</td>
<td>any</td>
</tr>
<tr>
<td></td>
<td>attempt to open disc file or $RECEIVE with maximum number of concurrent</td>
<td></td>
</tr>
<tr>
<td></td>
<td>operations greater than 1</td>
<td></td>
</tr>
<tr>
<td>29 (%35)</td>
<td>missing parameter</td>
<td>any</td>
</tr>
</tbody>
</table>
## APPENDIX B: FILE SYSTEM ERROR SUMMARY

### FILE SYSTEM ERROR SUMMARY (cont'd)

<table>
<thead>
<tr>
<th>Error Number</th>
<th>Description</th>
<th>Device Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>30 (%36)</td>
<td>unable to obtain main memory space for a link control block</td>
<td>any except 2</td>
</tr>
<tr>
<td>31 (%37)</td>
<td>unable to obtain SHORTPOOL space for a file system buffer area (I only); unable to obtain file system buffer space (II only)</td>
<td>any</td>
</tr>
<tr>
<td>32 (%40)</td>
<td>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 &lt;file number&gt; = -1 but no file was open (both systems)</td>
<td>any</td>
</tr>
<tr>
<td>33 (%41)</td>
<td>i/o process unable to obtain IOPOOL space for i/o buffer, or count too large for dedicated i/o buffer (I only); i/o process unable to obtain i/o segment space (II only) read from unstructured disc spans too many sectors</td>
<td>any except 2</td>
</tr>
<tr>
<td>34 (%42)</td>
<td>unable to obtain file system control block (II only)</td>
<td>any</td>
</tr>
<tr>
<td>35 (%43)</td>
<td>unable to obtain i/o process control block (II only)</td>
<td>any except 2</td>
</tr>
<tr>
<td>36 (%44)</td>
<td>unable to obtain physical memory (II only)</td>
<td>any</td>
</tr>
<tr>
<td>37 (%45)</td>
<td>unable to obtain physical memory for i/o (II only)</td>
<td>any except 2</td>
</tr>
<tr>
<td>38 (%46)</td>
<td>operation attempted on wrong type of system</td>
<td>any except 2</td>
</tr>
<tr>
<td>40 (%50)</td>
<td>operation timed out</td>
<td>any</td>
</tr>
<tr>
<td>41 (%51)</td>
<td>checksum error on file synchronization block</td>
<td>3</td>
</tr>
<tr>
<td>Error Number</td>
<td>Description</td>
<td>Device Type</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>42 (%52)</td>
<td>attempt to read from unallocated extent</td>
<td>3</td>
</tr>
<tr>
<td>43 (%53)</td>
<td>unable to obtain disc space for extent</td>
<td>3</td>
</tr>
<tr>
<td>44 (%54)</td>
<td>directory is full</td>
<td>3</td>
</tr>
<tr>
<td>45 (%55)</td>
<td>file is full</td>
<td>3</td>
</tr>
<tr>
<td>46 (%56)</td>
<td>invalid key specified</td>
<td>3</td>
</tr>
<tr>
<td>47 (%57)</td>
<td>key not consistent with file data</td>
<td>3</td>
</tr>
<tr>
<td>48 (%60)</td>
<td>security violation, or remote password illegal or does not exist</td>
<td>3</td>
</tr>
<tr>
<td>49 (%61)</td>
<td>access violation</td>
<td>any except 2</td>
</tr>
<tr>
<td>50 (%62)</td>
<td>directory error</td>
<td>3</td>
</tr>
<tr>
<td>51 (%63)</td>
<td>directory is bad</td>
<td>3</td>
</tr>
<tr>
<td>52 (%64)</td>
<td>error in disc free space table</td>
<td>3</td>
</tr>
<tr>
<td>53 (%65)</td>
<td>file system internal error</td>
<td>3</td>
</tr>
<tr>
<td>54 (%66)</td>
<td>i/o error in disc free space table</td>
<td>3</td>
</tr>
<tr>
<td>55 (%67)</td>
<td>i/o error in directory</td>
<td>3</td>
</tr>
<tr>
<td>56 (%70)</td>
<td>i/o error on volume label</td>
<td>3</td>
</tr>
<tr>
<td>57 (%71)</td>
<td>disc free space table is full</td>
<td>3</td>
</tr>
<tr>
<td>58 (%72)</td>
<td>disc free space table is bad</td>
<td>3</td>
</tr>
<tr>
<td>59 (%73)</td>
<td>file is bad</td>
<td>3</td>
</tr>
<tr>
<td>60 (%74)</td>
<td>volume on which this file resides has been removed, device has been downed, or process has failed since the file was opened</td>
<td>any except 1 and 2</td>
</tr>
<tr>
<td>Error Number</td>
<td>Description</td>
<td>Device Type</td>
</tr>
<tr>
<td>-------------</td>
<td>------------------------------------------------------------------------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>61 (%75)</td>
<td>no more file opens permitted on this volume</td>
<td>3</td>
</tr>
<tr>
<td>62 (%76)</td>
<td>volume has been mounted, but mount order has not been given</td>
<td>3</td>
</tr>
<tr>
<td>63 (%77)</td>
<td>volume has been mounted and mount is in progress (waiting for mount interrupt)</td>
<td>3</td>
</tr>
<tr>
<td>64 (%100)</td>
<td>volume has been mounted and mount is in progress</td>
<td>3</td>
</tr>
<tr>
<td>65 (%101)</td>
<td>only special requests permitted</td>
<td>3</td>
</tr>
<tr>
<td>66 (%102)</td>
<td>device has been downed by operator, or hard failure occurred on controller</td>
<td>any except 2</td>
</tr>
<tr>
<td>70 (%106)</td>
<td>continue file operation</td>
<td>0, 3</td>
</tr>
<tr>
<td>71 (%107)</td>
<td>duplicate record</td>
<td>3</td>
</tr>
<tr>
<td>72 (%110)</td>
<td>access to secondary partition not permitted</td>
<td>3</td>
</tr>
<tr>
<td>73 (%111)</td>
<td>file or record locked</td>
<td>3</td>
</tr>
<tr>
<td>74 (%112)</td>
<td>READUPDATE called for $RECEIVE and number of messages queued exceeds receive depth, ( \text{REPLY} ) called with an invalid message tag, or ( \text{REPLY} ) called with no message outstanding</td>
<td>2</td>
</tr>
<tr>
<td>75 (%113)</td>
<td>requesting process has no current process TRANSID</td>
<td>3</td>
</tr>
<tr>
<td>76 (%114)</td>
<td>transaction is in the process of ending</td>
<td>3 or none</td>
</tr>
<tr>
<td>77 (%115)</td>
<td>a TMF system file has the wrong file code</td>
<td>3</td>
</tr>
<tr>
<td>78 (%116)</td>
<td>TRANSID is invalid or obsolete</td>
<td>3 or none</td>
</tr>
<tr>
<td>79 (%117)</td>
<td>attempt made by TRANSID to update or delete a record it has not previously locked</td>
<td>3</td>
</tr>
</tbody>
</table>
## FILE SYSTEM ERROR SUMMARY (cont'd)

<table>
<thead>
<tr>
<th>Error Number</th>
<th>Description</th>
<th>Device Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>80 (%120)</td>
<td>invalid operation attempted on audited file or non-audited disc volume</td>
<td>3</td>
</tr>
<tr>
<td>81 (%121)</td>
<td>attempted operation invalid for TRANSID that has no-wait i/o outstanding on a disc or process file</td>
<td>2 or none</td>
</tr>
<tr>
<td>82 (%122)</td>
<td>TMF is not running</td>
<td>0, 3, or none</td>
</tr>
<tr>
<td>83 (%123)</td>
<td>process has initiated more concurrent transactions than can be handled</td>
<td>none</td>
</tr>
<tr>
<td>84 (%124)</td>
<td>TMF is not configured</td>
<td>0, 3, or none</td>
</tr>
<tr>
<td>87 (%127)</td>
<td>waiting on a READ request and did not get it</td>
<td>10</td>
</tr>
<tr>
<td>88 (%130)</td>
<td>a CONTROL READ is pending; new READ invalid</td>
<td>10</td>
</tr>
<tr>
<td>89 (%131)</td>
<td>remote device has no buffer available</td>
<td>10</td>
</tr>
<tr>
<td>90 (%132)</td>
<td>TRANSID aborted because its parent process died</td>
<td>3 or none</td>
</tr>
<tr>
<td>91 (%133)</td>
<td>Internal software error</td>
<td>4.2</td>
</tr>
<tr>
<td>92 (%134)</td>
<td>TRANSID aborted because path to remote node is down</td>
<td>3 or none</td>
</tr>
<tr>
<td>93 (%135)</td>
<td>TRANSID aborted because it spanned too many audit files</td>
<td>3 or none</td>
</tr>
<tr>
<td>94 (%136)</td>
<td>TRANSID aborted by operator command</td>
<td>3 or none</td>
</tr>
<tr>
<td>97 (%141)</td>
<td>TRANSID was aborted</td>
<td>3 or none</td>
</tr>
<tr>
<td>98 (%142)</td>
<td>Transaction Monitor Process's Network Active Transactions table is full</td>
<td>0, 3, or none</td>
</tr>
<tr>
<td>Error Number</td>
<td>Description</td>
<td>Device Type</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>99 (%143)</td>
<td>attempt to use microcode option that is not installed</td>
<td>any except 2</td>
</tr>
<tr>
<td>100 (%144)</td>
<td>device not ready</td>
<td>any except 2</td>
</tr>
<tr>
<td>101 (%145)</td>
<td>no write ring</td>
<td>4</td>
</tr>
<tr>
<td>102 (%146)</td>
<td>paper out or bail not properly closed</td>
<td>5</td>
</tr>
<tr>
<td>103 (%147)</td>
<td>disc not ready due to power failure</td>
<td>3</td>
</tr>
<tr>
<td>104 (%150)</td>
<td>no response from device</td>
<td>5.4</td>
</tr>
<tr>
<td>105 (%151)</td>
<td>VFU error</td>
<td>5.4</td>
</tr>
<tr>
<td>110 (%156)</td>
<td>only break access permitted</td>
<td>6, 61</td>
</tr>
<tr>
<td>111 (%157)</td>
<td>operation aborted because of break</td>
<td>6, 61</td>
</tr>
<tr>
<td>112 (%160)</td>
<td>READ or WRITEREAD preempted by operator message</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>too many user console messages</td>
<td>1</td>
</tr>
<tr>
<td>120 (%170)</td>
<td>data parity error</td>
<td>any except 2</td>
</tr>
<tr>
<td>121 (%171)</td>
<td>data overrun error</td>
<td>any except 2</td>
</tr>
<tr>
<td>122 (%172)</td>
<td>request aborted due to possible data loss caused by reset of circuit</td>
<td>6, 9, 11, 61</td>
</tr>
<tr>
<td>123 (%173)</td>
<td>subdevice busy</td>
<td>5, 6, 10</td>
</tr>
<tr>
<td>124 (%174)</td>
<td>line reset is in progress</td>
<td>6, 10, 59, 60</td>
</tr>
<tr>
<td>130 (%202)</td>
<td>illegal address to disc</td>
<td>3</td>
</tr>
<tr>
<td>Error Number</td>
<td>Description</td>
<td>Device Type</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------------------------------------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>131 (%203)</td>
<td>write check error from disc</td>
<td>3.0, 3.1</td>
</tr>
<tr>
<td>132 (%204)</td>
<td>seek incomplete from disc</td>
<td>3.0, 3.1</td>
</tr>
<tr>
<td>133 (%205)</td>
<td>access not ready on disc</td>
<td>3.0, 3.1</td>
</tr>
<tr>
<td>134 (%206)</td>
<td>address compare error on disc</td>
<td>3</td>
</tr>
<tr>
<td>135 (%207)</td>
<td>write protect violation with disc</td>
<td>3</td>
</tr>
<tr>
<td>136 (%210)</td>
<td>unit ownership error (dual-port disc)</td>
<td>3</td>
</tr>
<tr>
<td>137 (%211)</td>
<td>controller buffer parity error</td>
<td>any except 2</td>
</tr>
<tr>
<td>138 (%212)</td>
<td>interrupt overrun</td>
<td>any except 2</td>
</tr>
<tr>
<td>139 (%213)</td>
<td>controller error</td>
<td>any except 2</td>
</tr>
<tr>
<td>140 (%214)</td>
<td>modem error, or modem or link disconnected</td>
<td>6, 7, 10, 11, 12, 59, 60, 61, 63</td>
</tr>
<tr>
<td>145 (%221)</td>
<td>card reader motion check error</td>
<td>8</td>
</tr>
<tr>
<td>146 (%222)</td>
<td>card reader read check error</td>
<td>8</td>
</tr>
<tr>
<td>147 (%223)</td>
<td>invalid Hollerith code read</td>
<td>8</td>
</tr>
<tr>
<td>150 (%226)</td>
<td>end-of-tape marker detected</td>
<td>4</td>
</tr>
<tr>
<td>151 (%227)</td>
<td>runaway tape detected</td>
<td>4</td>
</tr>
<tr>
<td>152 (%230)</td>
<td>unusual end -- unit went offline</td>
<td>4, 11</td>
</tr>
<tr>
<td>153 (%231)</td>
<td>tape drive power on</td>
<td>4</td>
</tr>
<tr>
<td>154 (%232)</td>
<td>BOT detected during backspace files or backspace records</td>
<td>4</td>
</tr>
<tr>
<td>155 (%233)</td>
<td>only nine-track tape permitted</td>
<td>4</td>
</tr>
<tr>
<td>Error Number</td>
<td>Description</td>
<td>Device Type</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>--------------</td>
</tr>
<tr>
<td>156 (%234)</td>
<td>TIL protocol violation detected</td>
<td>12</td>
</tr>
<tr>
<td>157 (%235)</td>
<td>i/o process internal error</td>
<td>any except 2</td>
</tr>
<tr>
<td>158 (%236)</td>
<td>invalid function requested for HyperLink</td>
<td>26</td>
</tr>
<tr>
<td>160 (%240)</td>
<td>request is invalid for line state</td>
<td>6, 7, 10, 11</td>
</tr>
<tr>
<td></td>
<td>more than 7 reads or 7 writes issued</td>
<td>11</td>
</tr>
<tr>
<td>161 (%241)</td>
<td>impossible event occurred for line state</td>
<td>7, 10, 11</td>
</tr>
<tr>
<td>162 (%242)</td>
<td>operation timed out</td>
<td>7, 10, 11</td>
</tr>
<tr>
<td>163 (%243)</td>
<td>EOT received</td>
<td>7.0-7.3,7.8</td>
</tr>
<tr>
<td></td>
<td>power at auto-call unit is off</td>
<td>7.56, 11</td>
</tr>
<tr>
<td>164 (%244)</td>
<td>disconnect received</td>
<td>7.0, 7.1, 10, 11, 61</td>
</tr>
<tr>
<td></td>
<td>data line is occupied (busy)</td>
<td>7.56, 11</td>
</tr>
<tr>
<td>165 (%245)</td>
<td>RVI received</td>
<td>7.0 - 7.3</td>
</tr>
<tr>
<td></td>
<td>data line not occupied after setting call request</td>
<td>7.56, 11</td>
</tr>
<tr>
<td>166 (%246)</td>
<td>ENQ received</td>
<td>7.0, 7.1, 7.3, 7.9</td>
</tr>
<tr>
<td></td>
<td>auto-call unit failed to set &quot;present next digit&quot;</td>
<td>7.56, 11</td>
</tr>
<tr>
<td>167 (%247)</td>
<td>EOT received on line bid/select</td>
<td>7.0, 7.1, 7.3, 7.8</td>
</tr>
<tr>
<td></td>
<td>&quot;data set status&quot; not set after dialing all digits</td>
<td>7.56, 11</td>
</tr>
</tbody>
</table>
### FILE SYSTEM ERROR SUMMARY (cont'd)

<table>
<thead>
<tr>
<th>Error Number</th>
<th>Description</th>
<th>Device Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>168 (%250)</td>
<td>NAK received on line bid/select auto-call unit failed to clear &quot;present next digit&quot; after &quot;digit present&quot; was set</td>
<td>7.0, 7.1, 7.3, 7.8</td>
</tr>
<tr>
<td>169 (%251)</td>
<td>WACK received on line bid/select auto-call unit set &quot;abandon call and retry&quot;</td>
<td>7.56, 11</td>
</tr>
<tr>
<td></td>
<td>station disabled or station not defined</td>
<td>11</td>
</tr>
<tr>
<td>170 (%252)</td>
<td>no ID sequence received during circuit assurance mode invalid MCW entry number on WRITE</td>
<td>7.0, 7.1</td>
</tr>
<tr>
<td>171 (%253)</td>
<td>no response received on bid/poll/select</td>
<td>7, 10, 11</td>
</tr>
<tr>
<td>172 (%254)</td>
<td>reply not proper for protocol</td>
<td>6, 7, 10, 11</td>
</tr>
<tr>
<td>173 (%255)</td>
<td>maximum allowable NAKs received (transmission error) invalid MCW on WRITE</td>
<td>6, 7, 10</td>
</tr>
<tr>
<td>174 (%256)</td>
<td>WACK received after select aborted transmitted frame</td>
<td>7.2, 7.3</td>
</tr>
<tr>
<td>175 (%257)</td>
<td>incorrect alternating ACK received command reject</td>
<td>7.0 – 7.3</td>
</tr>
<tr>
<td>176 (%260)</td>
<td>poll sequence ended with no responder</td>
<td>7.3, 7.8, 7.9, 11.40</td>
</tr>
<tr>
<td>177 (%261)</td>
<td>text overrun</td>
<td>7, 10, 11</td>
</tr>
</tbody>
</table>
### FILE SYSTEM ERROR SUMMARY (cont'd)

<table>
<thead>
<tr>
<th>Error Number</th>
<th>Description</th>
<th>Device Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>178 (%262)</td>
<td>no address list specified</td>
<td>7.2, 7.3, 7.8, 7.9, 11.40, 61</td>
</tr>
<tr>
<td>179 (%263)</td>
<td>application buffer is incorrect</td>
<td>10, 61</td>
</tr>
<tr>
<td></td>
<td>control request pending or autopoll active</td>
<td>11.40</td>
</tr>
<tr>
<td>180 (%264)</td>
<td>unknown device status received</td>
<td>6.6 - 6.10, 5.3, 5.4, 10</td>
</tr>
<tr>
<td>181 (%265)</td>
<td>status receipt currently enabled for subdevice</td>
<td>10</td>
</tr>
<tr>
<td>190 (%276)</td>
<td>invalid status received from device</td>
<td>any except 2</td>
</tr>
<tr>
<td>191 (%277)</td>
<td>device power on</td>
<td>5</td>
</tr>
<tr>
<td>192 (%300)</td>
<td>device is being exercised</td>
<td>3 - 6</td>
</tr>
<tr>
<td>193 (%301)</td>
<td>invalid or missing microcode files</td>
<td>4.2</td>
</tr>
<tr>
<td>200 (%310)</td>
<td>device is owned by alternate port</td>
<td>any except 2</td>
</tr>
<tr>
<td>201 (%311)</td>
<td>current path to the device is down</td>
<td>any except 0 and 2</td>
</tr>
<tr>
<td></td>
<td>attempt was made to write to a nonexistent process</td>
<td></td>
</tr>
<tr>
<td>210 (%322)</td>
<td>device ownership changed during operation</td>
<td>any except 2</td>
</tr>
<tr>
<td>211 (%323)</td>
<td>failure of cpu performing this operation</td>
<td>any</td>
</tr>
<tr>
<td>212 (%324)</td>
<td>EIO instruction failure (I only)</td>
<td>any except 2</td>
</tr>
<tr>
<td>213 (%325)</td>
<td>channel data parity error</td>
<td>any except 2</td>
</tr>
<tr>
<td>Error Number</td>
<td>Description</td>
<td>Device Type</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>---------------------------</td>
</tr>
<tr>
<td>214 (%326)</td>
<td>channel timeout</td>
<td>any except 2</td>
</tr>
<tr>
<td>215 (%327)</td>
<td>i/o attempted to absent memory page</td>
<td>any except 2</td>
</tr>
<tr>
<td>216 (%330)</td>
<td>map parity error during this i/o (I only); or memory access breakpoint during this i/o (II only)</td>
<td>any except 2</td>
</tr>
<tr>
<td>217 (%331)</td>
<td>memory parity error during this i/o</td>
<td>any except 2</td>
</tr>
<tr>
<td>218 (%332)</td>
<td>interrupt timeout</td>
<td>any except 2</td>
</tr>
<tr>
<td>219 (%333)</td>
<td>illegal device reconnection</td>
<td>any except 2</td>
</tr>
<tr>
<td>220 (%334)</td>
<td>protect violation</td>
<td>any except 2</td>
</tr>
<tr>
<td>221 (%335)</td>
<td>channel pad-in violation (I only); controller handshake violation (II only)</td>
<td>any except 2</td>
</tr>
<tr>
<td>222 (%336)</td>
<td>bad channel status from EIO instruction</td>
<td>any except 2</td>
</tr>
<tr>
<td>223 (%337)</td>
<td>bad channel status from IIO instruction</td>
<td>any except 2</td>
</tr>
<tr>
<td>224 (%340)</td>
<td>controller error (I only)</td>
<td>any except 2</td>
</tr>
<tr>
<td>225 (%341)</td>
<td>no unit or multiple units assigned to same unit number</td>
<td>any except 2</td>
</tr>
<tr>
<td>226 (%342)</td>
<td>controller busy error (I only)</td>
<td>any except 2</td>
</tr>
</tbody>
</table>
## FILE SYSTEM ERROR SUMMARY (cont'd)

<table>
<thead>
<tr>
<th>Error Number</th>
<th>Description</th>
<th>Device Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>230 (%346)</td>
<td>cpu power on during this operation</td>
<td>any except 2</td>
</tr>
<tr>
<td>231 (%347)</td>
<td>controller power on during this operation</td>
<td>any except 2</td>
</tr>
<tr>
<td>240 (%350)</td>
<td>network line handler error; operation not started</td>
<td>any except 2</td>
</tr>
<tr>
<td>241 (%351)</td>
<td>network error; operation not started</td>
<td>any except 2</td>
</tr>
<tr>
<td>248 (%370)</td>
<td>network line handler process failed while this request was outstanding</td>
<td>any except 2</td>
</tr>
<tr>
<td>249 (%371)</td>
<td>network failure occurred while this request was outstanding</td>
<td>any except 2</td>
</tr>
<tr>
<td>250 (%372)</td>
<td>all paths to the system are down</td>
<td>any except 2</td>
</tr>
<tr>
<td>251 (%373)</td>
<td>network protocol error occurred</td>
<td>any except 2</td>
</tr>
<tr>
<td>300-511</td>
<td>reserved for use by application processes</td>
<td></td>
</tr>
</tbody>
</table>
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:

  \[
  \begin{align*}
  \text{<sysmsg>} & = -2 \\
  \text{<sysmsg>[l]} & = \text{cpu}
  \end{align*}
  \]

  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

  \[
  \begin{align*}
  \text{<sysmsg>} & = -2 \\
  \text{<sysmsg>[l] FOR 3} & = $\text{<process name>} \\
  \text{<sysmsg>[4]} & = -1
  \end{align*}
  \]

  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.
APPENDIX C: SYSTEM MESSAGES

- CPU Up Message
  
  \[
  \begin{align*}
  \langle \text{sysmsg} \rangle &= -3 \\
  \langle \text{sysmsg} \rangle[1] &= \text{cpu}
  \end{align*}
  \]

  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:

  \[
  \begin{align*}
  \langle \text{sysmsg} \rangle &= -5 \\
  \langle \text{sysmsg} \rangle[1] \text{ FOR } 4 &= \text{process ID of deleted process}
  \end{align*}
  \]

  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.

  \[
  \begin{align*}
  \langle \text{sysmsg} \rangle &= -5 \\
  \langle \text{sysmsg} \rangle[1] \text{ FOR } 3 &= \$<\text{process name}> \text{ of deleted process [pair]} \\
  \langle \text{sysmsg} \rangle[4] &= -1
  \end{align*}
  \]

  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:

  \[
  \begin{align*}
  \langle \text{sysmsg} \rangle &= -6 \\
  \langle \text{sysmsg} \rangle[1] \text{ FOR } 4 &= \text{process ID of deleted process}
  \end{align*}
  \]

  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.

  \[
  \begin{align*}
  \langle \text{sysmsg} \rangle &= -6 \\
  \langle \text{sysmsg} \rangle[1] \text{ FOR } 3 &= \$<\text{process name}> \text{ of deleted process [pair]} \\
  \langle \text{sysmsg} \rangle[4] &= -1
  \end{align*}
  \]

  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.
APPENDIX C: SYSTEM MESSAGES

- **Change in Status of Network Nodes**
  
  `<sysmsg> = -8`<br>`<sysmsg>[1].<0:7> = system number`<br>`<sysmsg>[1].<8:15> = number of cpu's`<br>`<sysmsg>[2] = current processor status bitmask`<br>`<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 MONITORNEXT procedure.

- **SETTIME Message (NonStop II systems only)**

  `<sysmsg> = -10`<br>`<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`<br>`<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)**


  This message is received by a process when a call to the NEWPROCESSNOWAIT procedure is completed.

- **BREAK Received from Terminal**

  `<sysmsg> = -20`<br>`<sysmsg>[1] = logical device number, in binary, of device where break was typed`<br>`<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.
APPENDIX C: SYSTEM MESSAGES

- **Time Signal Message (NonStop II systems only)**

  \[
  \begin{align*}
  \text{<sysmsg>} &= -22 \\
  \text{<sysmsg>[1]} &= \text{<parameter1>} \text{ supplied to SIGNALTIMEOUT} \\
  &\quad \text{(if none supplied, 0)} \\
  \text{<sysmsg>[2] FOR 2} &= \text{<parameter2>} \text{ supplied to SIGNALTIMEOUT} \\
  &\quad \text{(if none supplied, 0)}
  \end{align*}
  \]

  This message is received if a timer set by a call to SIGNALTIMEOUT has timed out.

- **Memory Lock Completion Message (NonStop II systems only)**

  \[
  \begin{align*}
  \text{<sysmsg>} &= -23 \\
  \text{<sysmsg>[1]} &= \text{<parameter1>} \text{ supplied to LOCKMEMORY} \\
  &\quad \text{(if none supplied, 0)} \\
  \text{<sysmsg>[2] FOR 2} &= \text{<parameter2>} \text{ supplied to LOCKMEMORY} \\
  &\quad \text{(if none supplied, 0)}
  \end{align*}
  \]

  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)**

  \[
  \begin{align*}
  \text{<sysmsg>} &= -24 \\
  \text{<sysmsg>[1]} &= \text{<parameter1>} \text{ supplied to LOCKMEMORY} \\
  &\quad \text{(if none supplied, 0)} \\
  \text{<sysmsg>[2] FOR 2} &= \text{<parameter2>} \text{ supplied to LOCKMEMORY} \\
  &\quad \text{(if none supplied, 0)}
  \end{align*}
  \]

  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 RESETSSYNC) is possible only if the process has opened its $RECEIVE$ file with <flags>.<l> = 1:

- **Process OPEN Message**

  \[
  \begin{align*}
  \text{<sysmsg>} &= -30 \\
  \text{<sysmsg>[1]} &= \text{<flags>} \text{ parameter to caller's OPEN} \\
  \text{<sysmsg>[2]} &= \text{<sync or receive depth>} \text{ parameter to caller's OPEN} \\
  \text{<sysmsg>[3] FOR 4} &= 0 \text{ if normal open, process ID of primary process if an open by a backup process} \\
  \text{<sysmsg>[7]} &= 0 \text{ if normal open, file number of file if an open by a backup process} \\
  \text{<sysmsg>[8]} &= \text{process accessor ID of opener} \\
  \text{<sysmsg>[9] FOR 4} &= \text{optional 1st qualif name of named process or blanks} \\
  \text{<sysmsg>[13] FOR 4} &= \text{optional 2nd qualif name of named process or blanks}
  \end{align*}
  \]
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 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 ciAstartup;
BEGIN
  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 command.
    subvol [ 0:3 ], !
    dname [ 0:3 ];
  END;
  STRUCT outfile;
  BEGIN
    INT volume [ 0:3 ], ! [21] OUT parameter <file name> of RUN command.
    subvol [ 0:3 ], !
    dname [ 0:3 ];
  END;
  STRING param [ 0:n-1 ]; ! [33] <parameter string> of RUN 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 )
END; ! ciAstartupAmsg.
```

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 $\text{REPLY} = 70$. 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.
APPENDIX C: SYSTEM MESSAGES

The form of the assign message is:

```c
STRUCT ci^assign; ! assign message.
BEGIN
  INT msg^code; ! [0] -2
  STRUCT logicalunit; ! PARAMETERS TO ASSIGN COMMAND.
  BEGIN
    STRING programeflen, ! [1] length in bytes of name [0:31]
      programe[0:30],! <program unit> | *
    femenlen, ! [17] length in bytes of name [0:31]
      fename[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],
      file [0:3];
  END;
  ! createspec
    exclusionspec, ! [50] %00 if SHARED,
      %20 if EXCLUSIVE, ! corresponds
      %60 if PROTECTED. to flag
    accessspec, ! [51] %0000 if I-O,
      %2000 if INPUT,
      %4000 if OUTPUT. ! param of
    recordsize, ! [52] <record size>.
    blocksize; ! [53] <block size>.
  END;
```

The length of this message is 108 bytes.
APPENDIX C: SYSTEM MESSAGES

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 \texttt{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:

```c
STRUCT ci^param;
BEGIN
  INT msg^code,
  numparams;
  STRING parameters [ 0:1023 ];
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 than the RUN), a wakeup message will be ignored.
The form of the wakeup message is:

```c
STRUCT wakeupAmsg;
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

```c
STRUCT displayAmsg;
BEGIN
  INT msgcode;
  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:

```c
STRUCT logonAmsg;
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.
APPENDIX C: SYSTEM MESSAGES

The form of the reply to the logon message is:

```c
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:

```c
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:

```c
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 processcreationAmsg;
BEGIN
  INT msgcode, userid, cipri,
  ciinfile [ 0:11 ], cioutfile[ 0:11 ], progname [ 0:11 ],
  priority, processor;
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 processcreationAreply;
BEGIN
  INT replycode, progname [ 0:11 ], priority, processor;
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:

```c
STRUCT processcreation^reply;
BEGIN
  INT replycode;         ! [0] l = 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.

D-1
APPENDIX D: SOURCE FOR $SYSTEM.SYSTEM.GPLDEFS

LITERAL
SHARE = 0,
EXCLUSIVE = 1,
PROTECTED = 3;

OPEN^FILE FLAGS

OPEN"FILE FLAGS v 111111
0123456789012345
1111111111222222222233

DEFINE
!4567890123456789012345678901
ABORT^OPENERR = %B0000000000000000000000000000010D#,
ABORT^XFERERR = %B0000000000000000000000000000010D#,
PRINT^ERR^MSG = %B0000000000000000000000000000010D#,
AUTO^CREATE = %B000000000000000000000000000001000D#,
MUSTBENEW = %B0000000000000000000000000000010000D#,
PURGE^DATA = %B00000000000000000000000000000100000D#,
AUTO^TOF = %B000000000000000000000000000001000000D#,
NOWAIT = %B0000000000000000000000000000010000000D#,
BLOCKED = %B00000000000000000000000000000100000000D#,
VAR^FORMAT = %B000000000000000000000000000001000000000D#,
READ^TRIM = %B0000000000000000000000000000010000000000D#,
WRITE^TRIM = %B00000000000000000000000000000100000000000D#,
WRITE^FOLD = %B00000000000000000000000000000100000000000D#,
WRITE^PAD = %B00000000000000000000000000000100000000000D#,
CRLF^BREAK = %B00000000000000000000000000000100000000000D#;

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,
APPENDIX D: SOURCE FOR $SYSTEM\_SYSTEM\_GPLDEFS

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,
APPENDIX D: SOURCE FOR $SYSTEM.SYSTEM.GPLDEFS

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^BUFTOOMALL = 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^FILENOTOPE = 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
APPENDIX D: SOURCE FOR $SYSTEM.SYSTEM.GPLDEFS

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
    !
    RECLLEN,  ! (20) logical record length.
    BLKBUFLEN, ! (21) block length from ASSIGN,

    !
    !
    ! open options.

    OPENEXCLUSION, ! (22) exclusion bits to OPEN.
APPENDIX E: SEQUENTIAL I/O FILE CONTROL BLOCK FORMAT

OPENACCESS;  ! (23) access bits to OPEN.
!
! initializer group.
!
INT PUCB^POINTER,
SAMEFILELINK;  ! (24) 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&I.
OPENFLAGS2,  ! (33) flags parameters to OPEN^F&I.
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,
BLKXFERCNT,  ! (39) word address of block buffer.
!
BLKREADCNT =  ! (40) number of bytes to be transferred between device and target buffer.
BLKWRITECNT =  ! (40) number of bytes to be written from target buffer to device.
BLKCNTXFERRED,  ! (41) number of bytes transferred between device and target buffer.
!
BLKCNTREAD =  ! (41) number of bytes read into target buffer.
BLKCNTWRITTEN =  ! (41) number of bytes written from target buffer.
BLKNEXTREC,  ! (42) (byte address) While blocking/deblocking this is the address...
APPENDIX E: SEQUENTIAL I/O FILE CONTROL BLOCK FORMAT

USRBUFADDR,  ! (43) byte address of user buffer.
USRWRCNT,    ! (44) <write count> parameter of
USRRDRCNT,   ! (45) <max read count> parameter of
TFOLDELLEN,  ! (46) terminal write fold length
USRCNTRD =    ! (46) number of bytes read into user
TFOLDELLEN,  ! (46) (= physical record length).
PHYSXFERCNT, ! (47) transfer count value passed to
PHYSIOCNTXFERRED, ! (48) count transferred value
PHYSIOCNTWR = ! (48) count read value returned from
PHYSIOCNTXFERRED; ! (48) count written value returned

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,
BACKUPPID =    ! (42:45) Backup opener's
BLKNEXTREC,

APPENDIX E: SEQUENTIAL I/O FILE CONTROL BLOCK FORMAT

REPLYCODE = ! (46) $RECEIVE reply error code.
   TFOLDLEN;
   ! Misc Group.
   !
   INT COMMCTRL = ! (47)
   PHYSXFERCNT,
   OPRQUESTFCBADDR = ! (48) FCB of file for which operator
   PHYSIOCNTXFERRED, console messages are being
   ! displayed. (see NOERROR,
   ! prompt).
   OPRQSTCOUNT = ! (49) Count of number of operator
   LOGICALFILENAME, messages displayed. (see
   ! NOERROR, prompt).
   ERRFCBADDR [-1:-1] = ! (50) FCB address of file where
   LOGICALFILENAME; errors are to be reported.

   !
   INT PXCNT,
   PRCNT = PXCNT,
   PWCONT = PXCNT;

   !
   INT SYSMSGSl,
   SYSMSGS2,
   SYSMSGS3,
   SYSMSGS4;

   !
   INT SPAREl;
   !
   INT CHECKSUM;
   !
   END; ! FCB"TMPL.
   !
   -- BIT FIELDS.
   !
   -- ASSIGN BITS.
   !
   DEFINE
   FILENAMESUPPLD = <0>#, = FCB.OPTIONSl.FILENAMESUPPLD#,
   FCB"FILENAMESUPPLD
   ! PRIEXTSUPPLD = <1>#, = FCB.OPTIONSl.PRIEXTSUPPLD#,
   FCB"PRIEXTSUPPLD
   ! SECEXTSUPPLD = <2>#, = FCB.OPTIONSl.SECEXTSUPPLD#,
   FCB"SECEXTSUPPLD
   ! FCODESUPPLD = <3>#, = FCB.OPTIONSl.FCODESUPPLD#,
   FCB"FCODESUPPLD
   ! EXCLUSIONSUPPLD = <4>#, = FCB.OPTIONSl.EXCLUSIONSUPPLD#,
   FCB"EXCLUSIONSUPPLD

E-4
APPENDIX E: SEQUENTIAL I/O FILE CONTROL BLOCK FORMAT

! 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#;
APPENDIX E: SEQUENTIAL I/O FILE CONTROL BLOCK FORMAT

! 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.

LITERAL
0123456789012345
ALLOWED^OPENFLAGS1 = %B0000111111111111;
!
!
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
APPENDIX E: SEQUENTIAL I/O FILE CONTROL BLOCK FORMAT

\[
\begin{align*}
! 0 &= \text{NO} \\
! 1 &= \text{YES} \\
FCB^{\text{AUTOTOF}} &= \text{FCB.OPENFLAGS2.AUTOTOF} \\
\text{DEFINE} \\
NOWAITIO &= <8>#, \text{! open with no-wait depth of} \ 1. \\
! 0 &= \text{WAIT} \\
! 1 &= \text{NO-WAIT} \\
FCB^{\text{NOWAITIO}} &= \text{FCB.OPENFLAGS2.NOWAITIO} \\
\text{DEFINE} \\
BLOCKEDIO &= <7>#, \text{! blocked i/o.} \\
! 0 &= \text{NOT BLOCKED} \\
! 1 &= \text{BLOCKED} \\
FCB^{\text{BLOCKEDIO}} &= \text{FCB.OPENFLAGS2.BLOCKEDIO} \\
\text{DEFINE} \\
VARFORMAT &= <6>#, \text{! variable length records.} \\
! 0 &= \text{FIXED LENGTH} \\
! 1 &= \text{VARIABLE LENGTH} \\
FCB^{\text{VARFORMAT}} &= \text{FCB.OPENFLAGS2.VARFORMAT} \\
\text{DEFINE} \\
READTRIM &= <5>#, \text{! trim trailing blanks.} \\
! 0 &= \text{NOTRIM} \\
! 1 &= \text{TRIM} \\
FCB^{\text{READTRIM}} &= \text{FCB.OPENFLAGS2.READTRIM} \\
\text{DEFINE} \\
WRITETRIM &= <4>#, \text{! trim trailing blanks.} \\
! 0 &= \text{NOTRIM} \\
! 1 &= \text{TRIM} \\
FCB^{\text{WRITETRIM}} &= \text{FCB.OPENFLAGS2.WRITETRIM} \\
\text{DEFINE} \\
WRITEFOLD &= <3>#, \text{! fold write transfers greater than} \\
! 0 &= \text{TRUNCATE} \quad \text{write record length bytes into} \\
! 1 &= \text{FOLD} \quad \text{multiple records.} \\
FCB^{\text{WRITEFOLD}} &= \text{FCB.OPENFLAGS2.WRITEFOLD} \\
\text{DEFINE} \\
WRITEPAD &= <2>#, \text{! pad record with trailing blanks.} \\
FCB^{\text{WRITEPAD}} &= \text{FCB.OPENFLAGS2.WRITEPAD} \\
\text{DEFINE} \\
CRLFBREAK &= <1>#, \text{! carriage return/line feed on break.} \\
! 0 &= \text{NO CRLF ON BREAK} \\
! 1 &= \text{CRLF ON BREAK} \\
FCB^{\text{CRLFBREAK}} &= \text{FCB.OPENFLAGS2.CRLFBREAK} \\
\end{align*}
\]

! allowable open flags 2 settings.

\[
\begin{align*}
\text{LITERAL} \\
&111111 \\
\text{LITERAL} \\
&10123456789012345 \\
\text{ALLOWED}^{\text{OPENFLAGS2}} &= \text{%B1111111111111111}; \\
\text{LITERAL} \\
&111111 \\
\text{LITERAL} \\
&10123456789012345 \\
\text{DEFAULT}^{\text{OPENFLAGS2}} &= \text{%B010110001001111}; \\
\text{LITERAL} \\
&111111 \\
\text{TRANSFER CONTROL} ( \text{FCB.XFERCNTL} )
\end{align*}
\]
APPENDIX E: SEQUENTIAL I/O FILE CONTROL BLOCK FORMAT

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#;

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 )

E-8
APPENDIX E: SEQUENTIAL I/O FILE CONTROL BLOCK FORMAT

! -- 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 '>>'
! EDBUFSHIFT (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>#, ! last break mode from SETMODE.
FCB^CNTLINPROGRESS = FCB.PHYSXFERCNT.CNTLINPROGRESS#,
FORMSCNTOPL = <1:15>#, !$RECEIVE has been opened for data
! transfer.
FCB^FORMSCNTOPL = FCB.PHYSXFERCNT.FORMSCNTOPL#;

! 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 = COMMFCB.BRKFBCBADDR#;

! $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#,
RCVPENCT = <2:3>#, ! count of OPEN messages received.
COMMFCB^RCVPENCT = COMMFCB.RCVCNTL.RCVPENCT#,
APPENDIX E: SEQUENTIAL I/O FILE CONTROL BLOCK FORMAT

RCVSTATE = <4>,
! 0 = NEED READUPDATE.
! 1 = NEED REPLY.
COMMFCB^RCVSTATE = COMMFCB.RVCNTL.RCVSTATE#,
RCVUSEROPENREPLY = <5>, ! user will reply to OPEN messages.
! 0 = SIO REPLIES.
! 1 = USER REPLIES.
COMMFCB^RCVUSEROPENREPLY = COMMFCB.RVCNTL.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.RVCNTL.RCVPSUEDOEOF#,
MONCPUMSG = <2:3>, ! user CPU Up/Down messages.
COMMFCB^MONCPUMSG = COMMFCB.SYMSGS1.MONCPUMSG#,
OPENMSG = <14>, ! user wants OPEN messages.
COMMFCB^OPENMSG = COMMFCB.SYMSGS2.OPENMSG#,
CLOSEMSG = <15>, ! user wants CLOSE messages.
COMMFCB^CLOSEMSG = COMMFCB.SYMSGS2.CLOSEMSG#;

! COMMON CONTROL ( COMMFCB.COMMCNTL )

DEFINE
CREATEINPROGRESS = <0>, ! l during call to OPEN^FILE while
! creating.
COMMFCB^CREATEINPROGRESS = COMMFCB.COMMCNTL.CREATEINPROGRESS#,
OPENINPROGRESS = <1>, ! l 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>, ! l = 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:

\[
\begin{align*}
040400 \quad \text{("left byte" value of "A")} \\
+ 000102 \quad \text{("right byte" value of "B")} \\
\hline \\
040502 \quad \text{(total)}
\end{align*}
\]
## APPENDIX F: ASCII CHARACTER SET

<table>
<thead>
<tr>
<th>Character</th>
<th>Octal Value (left byte)</th>
<th>Octal Value (right byte)</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUL</td>
<td>000000</td>
<td>000000</td>
<td>Null</td>
</tr>
<tr>
<td>SOH</td>
<td>000400</td>
<td>000000</td>
<td>Start of heading</td>
</tr>
<tr>
<td>STX</td>
<td>001000</td>
<td>000002</td>
<td>Start of text</td>
</tr>
<tr>
<td>ETX</td>
<td>001400</td>
<td>000003</td>
<td>End of text</td>
</tr>
<tr>
<td>EOT</td>
<td>002000</td>
<td>000004</td>
<td>End of transmission</td>
</tr>
<tr>
<td>ENQ</td>
<td>002400</td>
<td>000005</td>
<td>Enquiry</td>
</tr>
<tr>
<td>ACK</td>
<td>003000</td>
<td>000006</td>
<td>Acknowledge</td>
</tr>
<tr>
<td>BEL</td>
<td>003400</td>
<td>000007</td>
<td>Bell</td>
</tr>
<tr>
<td>BS</td>
<td>004000</td>
<td>000010</td>
<td>Backspace</td>
</tr>
<tr>
<td>HT</td>
<td>004400</td>
<td>000011</td>
<td>Horizontal tabulation</td>
</tr>
<tr>
<td>LF</td>
<td>005000</td>
<td>000012</td>
<td>Line feed</td>
</tr>
<tr>
<td>VT</td>
<td>005400</td>
<td>000013</td>
<td>Vertical tabulation</td>
</tr>
<tr>
<td>FF</td>
<td>006000</td>
<td>000014</td>
<td>Form feed</td>
</tr>
<tr>
<td>CR</td>
<td>006400</td>
<td>000015</td>
<td>Carriage return</td>
</tr>
<tr>
<td>SO</td>
<td>007000</td>
<td>000016</td>
<td>Shift out</td>
</tr>
<tr>
<td>SI</td>
<td>007400</td>
<td>000017</td>
<td>Shift in</td>
</tr>
<tr>
<td>DLE</td>
<td>010000</td>
<td>000020</td>
<td>Data link escape</td>
</tr>
<tr>
<td>DC1</td>
<td>010400</td>
<td>000021</td>
<td>Device control 1</td>
</tr>
<tr>
<td>DC2</td>
<td>011000</td>
<td>000022</td>
<td>Device control 2</td>
</tr>
<tr>
<td>DC3</td>
<td>011400</td>
<td>000023</td>
<td>Device control 3</td>
</tr>
<tr>
<td>DC4</td>
<td>012000</td>
<td>000024</td>
<td>Device control 4</td>
</tr>
<tr>
<td>NAK</td>
<td>012400</td>
<td>000025</td>
<td>Negative acknowledge</td>
</tr>
<tr>
<td>SYN</td>
<td>013000</td>
<td>000026</td>
<td>Synchronous idle</td>
</tr>
<tr>
<td>ETB</td>
<td>013400</td>
<td>000027</td>
<td>End of transmission block</td>
</tr>
<tr>
<td>CAN</td>
<td>014000</td>
<td>000030</td>
<td>Cancel</td>
</tr>
<tr>
<td>EM</td>
<td>014400</td>
<td>000031</td>
<td>End of medium</td>
</tr>
<tr>
<td>SUB</td>
<td>015000</td>
<td>000032</td>
<td>Substitute</td>
</tr>
<tr>
<td>ESC</td>
<td>015400</td>
<td>000033</td>
<td>Escape</td>
</tr>
<tr>
<td>FS</td>
<td>016000</td>
<td>000034</td>
<td>File separator</td>
</tr>
<tr>
<td>GS</td>
<td>016400</td>
<td>000035</td>
<td>Group separator</td>
</tr>
<tr>
<td>RS</td>
<td>017000</td>
<td>000036</td>
<td>Record separator</td>
</tr>
<tr>
<td>US</td>
<td>017400</td>
<td>000037</td>
<td>Unit separator</td>
</tr>
<tr>
<td>SP</td>
<td>020000</td>
<td>000040</td>
<td>Space</td>
</tr>
<tr>
<td>!</td>
<td>020400</td>
<td>000041</td>
<td>Exclamation point</td>
</tr>
<tr>
<td>&quot;</td>
<td>021000</td>
<td>000042</td>
<td>Quotation mark</td>
</tr>
<tr>
<td>#</td>
<td>021400</td>
<td>000043</td>
<td>Number sign</td>
</tr>
<tr>
<td>$</td>
<td>022000</td>
<td>000044</td>
<td>Dollar sign</td>
</tr>
<tr>
<td>%</td>
<td>022400</td>
<td>000045</td>
<td>Percent sign</td>
</tr>
<tr>
<td>&amp;</td>
<td>023000</td>
<td>000046</td>
<td>Ampersand</td>
</tr>
<tr>
<td>'</td>
<td>023400</td>
<td>000047</td>
<td>Apostrophe</td>
</tr>
</tbody>
</table>
## APPENDIX F: ASCII CHARACTER SET

<table>
<thead>
<tr>
<th>Character</th>
<th>Octal Value (left byte)</th>
<th>Octal Value (right byte)</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
<td>024000</td>
<td>000050</td>
<td>Opening parenthesis</td>
</tr>
<tr>
<td>)</td>
<td>024400</td>
<td>000051</td>
<td>Closing parenthesis</td>
</tr>
<tr>
<td>*</td>
<td>025000</td>
<td>000052</td>
<td>Asterisk</td>
</tr>
<tr>
<td>+</td>
<td>025400</td>
<td>000053</td>
<td>Plus</td>
</tr>
<tr>
<td>,</td>
<td>026000</td>
<td>000054</td>
<td>Comma</td>
</tr>
<tr>
<td>-</td>
<td>026400</td>
<td>000055</td>
<td>Hyphen (minus)</td>
</tr>
<tr>
<td>.</td>
<td>027000</td>
<td>000056</td>
<td>Period (decimal point)</td>
</tr>
<tr>
<td>/</td>
<td>027400</td>
<td>000057</td>
<td>Right slant</td>
</tr>
<tr>
<td>0</td>
<td>030000</td>
<td>000060</td>
<td>Zero</td>
</tr>
<tr>
<td>1</td>
<td>030400</td>
<td>000061</td>
<td>One</td>
</tr>
<tr>
<td>2</td>
<td>031000</td>
<td>000062</td>
<td>Two</td>
</tr>
<tr>
<td>3</td>
<td>031400</td>
<td>000063</td>
<td>Three</td>
</tr>
<tr>
<td>4</td>
<td>032000</td>
<td>000064</td>
<td>Four</td>
</tr>
<tr>
<td>5</td>
<td>032400</td>
<td>000065</td>
<td>Five</td>
</tr>
<tr>
<td>6</td>
<td>033000</td>
<td>000066</td>
<td>Six</td>
</tr>
<tr>
<td>7</td>
<td>033400</td>
<td>000067</td>
<td>Seven</td>
</tr>
<tr>
<td>8</td>
<td>034000</td>
<td>000070</td>
<td>Eight</td>
</tr>
<tr>
<td>9</td>
<td>034400</td>
<td>000071</td>
<td>Nine</td>
</tr>
<tr>
<td>:</td>
<td>035000</td>
<td>000072</td>
<td>Colon</td>
</tr>
<tr>
<td>;</td>
<td>035400</td>
<td>000073</td>
<td>Semicolon</td>
</tr>
<tr>
<td>&lt;</td>
<td>036000</td>
<td>000074</td>
<td>Less than</td>
</tr>
<tr>
<td>=</td>
<td>036400</td>
<td>000075</td>
<td>Equals</td>
</tr>
<tr>
<td>&gt;</td>
<td>037000</td>
<td>000076</td>
<td>Greater than</td>
</tr>
<tr>
<td>?</td>
<td>037400</td>
<td>000077</td>
<td>Question mark</td>
</tr>
<tr>
<td>@</td>
<td>040000</td>
<td>000100</td>
<td>Commercial at</td>
</tr>
<tr>
<td>A</td>
<td>040400</td>
<td>000101</td>
<td>Upper-case A</td>
</tr>
<tr>
<td>B</td>
<td>041000</td>
<td>000102</td>
<td>Upper-case B</td>
</tr>
<tr>
<td>C</td>
<td>041400</td>
<td>000103</td>
<td>Upper-case C</td>
</tr>
<tr>
<td>D</td>
<td>042000</td>
<td>000104</td>
<td>Upper-case D</td>
</tr>
<tr>
<td>E</td>
<td>042400</td>
<td>000105</td>
<td>Upper-case E</td>
</tr>
<tr>
<td>F</td>
<td>043000</td>
<td>000106</td>
<td>Upper-case F</td>
</tr>
<tr>
<td>G</td>
<td>043400</td>
<td>000107</td>
<td>Upper-case G</td>
</tr>
<tr>
<td>H</td>
<td>044000</td>
<td>000110</td>
<td>Upper-case H</td>
</tr>
<tr>
<td>I</td>
<td>044400</td>
<td>000111</td>
<td>Upper-case I</td>
</tr>
<tr>
<td>J</td>
<td>045000</td>
<td>000112</td>
<td>Upper-case J</td>
</tr>
<tr>
<td>K</td>
<td>045400</td>
<td>000113</td>
<td>Upper-case K</td>
</tr>
<tr>
<td>L</td>
<td>046000</td>
<td>000114</td>
<td>Upper-case L</td>
</tr>
<tr>
<td>M</td>
<td>046400</td>
<td>000115</td>
<td>Upper-case M</td>
</tr>
<tr>
<td>N</td>
<td>047000</td>
<td>000116</td>
<td>Upper-case N</td>
</tr>
<tr>
<td>O</td>
<td>047400</td>
<td>000117</td>
<td>Upper-case O</td>
</tr>
</tbody>
</table>
## APPENDIX F: ASCII CHARACTER SET

<table>
<thead>
<tr>
<th>Character</th>
<th>Octal Value (left byte)</th>
<th>Octal Value (right byte)</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>050000</td>
<td>000120</td>
<td>Upper-case P</td>
</tr>
<tr>
<td>Q</td>
<td>050400</td>
<td>000121</td>
<td>Upper-case Q</td>
</tr>
<tr>
<td>R</td>
<td>051000</td>
<td>000122</td>
<td>Upper-case R</td>
</tr>
<tr>
<td>S</td>
<td>051400</td>
<td>000123</td>
<td>Upper-case S</td>
</tr>
<tr>
<td>T</td>
<td>052000</td>
<td>000124</td>
<td>Upper-case T</td>
</tr>
<tr>
<td>U</td>
<td>052400</td>
<td>000125</td>
<td>Upper-case U</td>
</tr>
<tr>
<td>V</td>
<td>053000</td>
<td>000126</td>
<td>Upper-case V</td>
</tr>
<tr>
<td>W</td>
<td>053400</td>
<td>000127</td>
<td>Upper-case W</td>
</tr>
<tr>
<td>X</td>
<td>054000</td>
<td>000130</td>
<td>Upper-case X</td>
</tr>
<tr>
<td>Y</td>
<td>054400</td>
<td>000131</td>
<td>Upper-case Y</td>
</tr>
<tr>
<td>Z</td>
<td>055000</td>
<td>000132</td>
<td>Upper-case Z</td>
</tr>
<tr>
<td>[</td>
<td>055400</td>
<td>000133</td>
<td>Opening bracket</td>
</tr>
<tr>
<td>\</td>
<td>056000</td>
<td>000134</td>
<td>Left slant</td>
</tr>
<tr>
<td>]</td>
<td>056400</td>
<td>000135</td>
<td>Closing bracket</td>
</tr>
<tr>
<td>^</td>
<td>057000</td>
<td>000136</td>
<td>Circumflex</td>
</tr>
<tr>
<td>_</td>
<td>057400</td>
<td>000137</td>
<td>Underscore</td>
</tr>
<tr>
<td>`</td>
<td>060000</td>
<td>000140</td>
<td>Grave accent</td>
</tr>
<tr>
<td>a</td>
<td>060400</td>
<td>000141</td>
<td>Lower-case a</td>
</tr>
<tr>
<td>b</td>
<td>061000</td>
<td>000142</td>
<td>Lower-case b</td>
</tr>
<tr>
<td>c</td>
<td>061400</td>
<td>000143</td>
<td>Lower-case c</td>
</tr>
<tr>
<td>d</td>
<td>062000</td>
<td>000144</td>
<td>Lower-case d</td>
</tr>
<tr>
<td>e</td>
<td>062400</td>
<td>000145</td>
<td>Lower-case e</td>
</tr>
<tr>
<td>f</td>
<td>063000</td>
<td>000146</td>
<td>Lower-case f</td>
</tr>
<tr>
<td>g</td>
<td>063400</td>
<td>000147</td>
<td>Lower-case g</td>
</tr>
<tr>
<td>h</td>
<td>064000</td>
<td>000150</td>
<td>Lower-case h</td>
</tr>
<tr>
<td>i</td>
<td>064400</td>
<td>000151</td>
<td>Lower-case i</td>
</tr>
<tr>
<td>j</td>
<td>065000</td>
<td>000152</td>
<td>Lower-case j</td>
</tr>
<tr>
<td>k</td>
<td>065400</td>
<td>000153</td>
<td>Lower-case k</td>
</tr>
<tr>
<td>l</td>
<td>066000</td>
<td>000154</td>
<td>Lower-case l</td>
</tr>
<tr>
<td>m</td>
<td>066400</td>
<td>000155</td>
<td>Lower-case m</td>
</tr>
<tr>
<td>n</td>
<td>067000</td>
<td>000156</td>
<td>Lower-case n</td>
</tr>
<tr>
<td>o</td>
<td>067400</td>
<td>000157</td>
<td>Lower-case o</td>
</tr>
<tr>
<td>p</td>
<td>070000</td>
<td>000160</td>
<td>Lower-case p</td>
</tr>
<tr>
<td>q</td>
<td>070400</td>
<td>000161</td>
<td>Lower-case q</td>
</tr>
<tr>
<td>r</td>
<td>071000</td>
<td>000162</td>
<td>Lower-case r</td>
</tr>
<tr>
<td>s</td>
<td>071400</td>
<td>000163</td>
<td>Lower-case s</td>
</tr>
<tr>
<td>t</td>
<td>072000</td>
<td>000164</td>
<td>Lower-case t</td>
</tr>
<tr>
<td>u</td>
<td>072400</td>
<td>000165</td>
<td>Lower-case u</td>
</tr>
<tr>
<td>v</td>
<td>073000</td>
<td>000166</td>
<td>Lower-case v</td>
</tr>
<tr>
<td>w</td>
<td>073400</td>
<td>000167</td>
<td>Lower-case w</td>
</tr>
</tbody>
</table>
## APPENDIX F: ASCII CHARACTER SET

<table>
<thead>
<tr>
<th>Character</th>
<th>Octal Value (left byte)</th>
<th>Octal Value (right byte)</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>074000</td>
<td>000170</td>
<td>Lower-case x</td>
</tr>
<tr>
<td>y</td>
<td>074400</td>
<td>000171</td>
<td>Lower-case y</td>
</tr>
<tr>
<td>z</td>
<td>075000</td>
<td>000172</td>
<td>Lower-case z</td>
</tr>
<tr>
<td>{</td>
<td>075400</td>
<td>000173</td>
<td>Opening brace</td>
</tr>
<tr>
<td>}</td>
<td>076000</td>
<td>000174</td>
<td>Vertical line</td>
</tr>
<tr>
<td>~</td>
<td>076400</td>
<td>000175</td>
<td>Closing brace</td>
</tr>
<tr>
<td>DEL</td>
<td>077000</td>
<td>000176</td>
<td>Tilde</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Delete</td>
</tr>
</tbody>
</table>
INDEX

ABEND procedure 3.2-3
Access Control Block (ACB) 2.1-22
Access coordination 2.1-11
Accessing card readers 2.8-4
Accessing line printers 2.6-2
Accessing tape units 2.7-3
Accessing terminals 2.5-4
termination when reading 2.5-5
ACTIVATEPROCESS procedure 3.2-4
Active state, of a process 3.1-6
ADDUSER command 7.1-10, 7.1-12
Advanced checkpointing 5.4-1
Advanced file system 2.11-1
Advanced memory management 8.2-1
ALLOCATESEGMENT procedure 8.1-4
ALTERPRIORITY procedure 3.2-5
Ancestor process 3.1-13
ARMTRAP procedure 6-4
ASCII character set F-1
ASSIGN command 11-17
assign message 11-20
Attributes summary
  FCB 9-64
AUTOANSWER mode for 5508 printer 2.6-6
AWAITIO procedure 2.3-7
Backup process 1-8, 2.9-10, 3.1-12, 5.1-1, 5.3-1
Break feature 2.5-25
  break mode 2.5-29
  BREAK system message 2.5-26
  using BREAK (multiple processes) 2.5-28
  using BREAK (single process) 2.5-26
Buffering
  i/o system 2.1-26
  resident 2.11-5
CANCEL procedure 2.3-11
CANCELREQ procedure 2.3-12
CANCELTIMEOUT procedure 3.2-6
INDEX

Card readers 2.8-1
  accessing 2.8-4
  applicable procedures 2.8-1
  characteristics 2.8-1
  error recovery 2.8-5
  read modes 2.8-2
    ASCII 2.8-2
    column-binary 2.8-3
    packed-binary 2.8-4
CHECKCLOSE procedure 5.2-3
CHECKMONITOR procedure 5.2-5
  actions 5.2-7
CHECKOPEN procedure 5.2-9
CHECKPOINT procedure 5.2-12
Checkpointing 5.3-14
  action for CHECKPOINT failure 5.3-21
  advanced checkpointing 5.4-1
    backup open 5.4-1
      file synchronization information 5.4-2
  considerations for no-wait i/o 5.3-21
  creating a descendant process (pair) 5.3-28
  guidelines for checkpointing 5.3-15
  multiple disc updates 5.3-21
  opening a file during processing 5.3-27
  system messages 5.3-22
  takeover by backup 5.3-25
Checkpointing facility 1-14, 5.1-1
  data buffers 5.1-5
  data stack 5.1-5
NonStop overview 5.1-2
  sync blocks 5.1-5
  using the checkpointing facility 5.3-1
    file opening 5.3-13
    main processing loop 5.3-13
    startup for named process pairs 5.3-1
    startup for non-named process pairs 5.3-9
Checkpointing procedures 5.1-1, 5.2-1
  CHECKCLOSE 5.2-3
  CHECKMONITOR 5.2-5
  CHECKOPEN 5.2-9
  CHECKPOINT 5.2-12
  CHECKPOINTMANY 5.2-14
  CHECKSWITCH 5.2-17
  GETSYNCINFO 5.2-18
  MONITORCPUS 5.2-19
    PROCESSORSTATUS 5.2-21
  RESESYNC 5.2-22
  SETSYNCINFO 5.2-23
CHECKPOINTMANY procedure 5.2-14
Checksum processing 2.5-22
CHECKSWITCH procedure 5.2-17
CHECK\textasciitilde BREAK procedure 9-4

Index-2
INDEX

CHECKAFILE procedure 9-5
  example 9-11
  file types 9-7
  operations 9-5
CLEAR command 11-25
CLOSE procedure 2.3-13
CLOSEAFILE procedure 9-12
Closing a file 2.1-27
Command Interpreter 1-18, 11-1
Command Interpreter/program interface 11-1
Commands, Command Interpreter
  ADDUSER 7.1-10, 7.1-12
  ASSIGN 11-17
  CLEAR 11-25
  DEFAULT 7.1-10, 7.1-12
  DELUSER 7.1-10, 7.1-12
  LOGOFF 7.1-10
  LOGON 7.1-10
  PARAM 11-22
  PASSWORD 7.1-10, 7.1-12
  REMOTEPASSWORD 7.1-10
  RUN 11-12
  USERS 7.1-10, 7.1-12
  VOLUME 7.1-10, 7.1-12
Commands, FUP
  GIVE 7.1-10, 7.1-13
  INFO 7.1-10, 7.1-13
  LICENSE 7.1-10, 7.1-13
  REVOKE 7.1-10, 7.1-13
  SECURE 7.1-10, 7.1-13
Communicating with a new process 3.3-1
CONTIME procedure 4-2
CONTROL procedure 2.3-15
CONTROLBUF for 5520 printer 2.6-10
CONTROLBUF procedure 2.3-20
Conversion modes, 7-track tape
  ASCIIBC 2.7-18
  BINARYlTOl 2.7-23
  BINARY2TO3 2.7-22
  BINARY3TO4 2.7-21
  selecting a conversion mode 2.7-23
CONVERTPROCESSNAME procedure 3.2-7
CREATE procedure 2.3-23
CREATEPROCESSNAME procedure 3.2-8
CREATEREMOTENAME procedure 3.2-10
Creating a new process 1-5, 3.1-5, 3.3-1
Creator 1-5, 3.1-9
Creator accessor ID 7.1-5
CREATORACCESSID procedure 7.2-2
CTRLANSWER mode for 5508 printer 2.6-6

DAVFU 2.6-7
DEALLOCATESEGMENT procedure 8.1-6
Debug facility 1-10

Index-3
DEBUG procedure 4-3
Decorations, formatter
condition specifiers
M, minus 10-44
N, null 10-44
O, overflow 10-44
P, plus 10-44
Z, zero 10-44
location specifiers
A, absolute 10-44
F, floating 10-44
P, prior 10-44
DEFAULT command 7.1-10, 7.1-12
Default security for disc files 7.1-7
Default system 11-9
Default volume and subvolume 11-4
DEFINEPOOL procedure 8.1-7
DELAY procedure 3.2-11
Deleting a process 1-6, 3.1-7
DELUSER command 7.1-10, 7.1-12
Descriptors, formatter
   non-repeatable edit descriptors 10-20
   repeatable edit descriptors 10-26
Device names 2.2-3, 2.3-52
Device numbers, logical 2.2-3
Device types and subtypes 2.3-27
DEVICEINFO procedure 2.3-26
Disc error recovery
   path errors 2.1-28
Disc file security 7.1-6
   adopting owner ID 7.1-8
   determining access permission 7.1-7
Disc files 2.1-1
Display message 11-28
Echo 2.5-22
Edit descriptors, formatter 10-17
"A", data transfer 10-26
"P", data transfer 10-28
"E", data transfer 10-28
"F", data transfer 10-31
"G", data transfer 10-32
"I", data transfer 10-34
"L", data transfer 10-35
"M", data transfer 10-37
"",", quotation marks, literal 10-21
"'", apostrophes, literal 10-21
/, buffer control 10-24
:\, buffer control 10-24
BN, blank interpretation control 10-24
BZ, blank interpretation control 10-24
H, Hollerith 10-21
P, implied decimal point 10-22
S, optional plus control 10-23
INDEX

SP, optional plus control 10-23
SS, optional plus control 10-23
T, tab absolute 10-20
TL, tab left 10-20
TR, tab right 10-20
X, tab right 10-20

Edit files 2.3-30, 2.3-34
EDITREAD procedure 2.3-30
EDITREADINIT procedure 2.3-34
Elapsed timeout 3.1-19
Error indication 2.1-35, 2.4-1
Error recovery 2.1-37, 2.4-28

Errors
5520 2.6-12
file system 2.4-1, B-1
FORMATDATA procedure 10-9
NEWPROCESS and NEWPROCESSNOWAIT 3.2-32
sequential i/o procedures 9-38

Example NonStop program 12.1-1
Executing a process 3.1-6
Execution priority 3.1-7, 3.4-1
Extended memory segments 8.1-1
space management within 8.1-3
External declarations 1-18
sequential i/o D-1

FCB Attributes
summary 9-64
File access 2.1-7
disc files 2.1-8
processes 2.1-10
terminals 2.1-10

File Control Block (FCB), in file system 2.1-22

File Control Block (FCB), in sequential
i/o procedures 9-41, E-1

File management procedures
RESERVELCBS 2.11-3

File names 2.2-1, 11-2
$0 2.2-6
$RECEIVE 2.2-3
default volume and subvolume 11-4
device names 2.2-3
disc file names 2.2-2
external form 2.2-1, 11-2
file name expansion 11-4
internal form 2.2-1, 11-2

logical device numbers 2.2-3
network file names 2.2-7
process ID 2.2-4
network form 2.2-8
obtaining a process ID 2.2-5
process name form 2.2-4
timestamp form 2.2-4

Index-5
INDEX

File security checking 2.3-70
  disc files 7.1-6
File System 1/10
file system 1/10
File system 1-10, 2.1-1
  advanced features 2.11-1
File system errors 2.4-1, B-1
  error categories 2.4-1
  error recovery 2.4-29
File system implementation 2.1-16
  automatic disc path error recovery 2.1-28
  buffering 2.1-26
  file and i/o system structure 2.1-16
  file closing 2.1-27
  file opening 2.1-21
  file system procedure execution 2.1-21
  file transfers 2.1-24
  mirror volumes 2.1-34
File system procedures 2.11-1, 2.3-1
  <buffer> parameter 2.3-5
  <file number> parameters 2.3-4
  <tag> parameters 2.3-5
  <transfer count> parameter 2.3-5
  access mode (disc files) 2.3-6
  AWAITIO 2.3-7
  CANCEL 2.3-11
  CANCELREQ 2.3-12
  characteristics 2.3-3
  CLOSE 2.3-13
  completion 2.3-4
  condition codes 2.3-6
  CONTROL 2.3-15
  CONTROLBUF 2.3-20
  CREATE 2.3-23
  DEVICEINFO 2.3-26
  EDITREAD 2.3-30
  EDITREADINIT 2.3-34
  errors 2.3-6, 2.4-1
  FILEERROR 2.3-36
  FILEINFO 2.3-39
  FNAMECOLLAPSE 2.3-43
  FNAMECOMPARE 2.3-45
  FNAMEEXPAND 2.3-48
  GETDEVNAME 2.3-52
  GETSYSTEMNAME 2.3-54
  LASTRECEIVE 2.3-55
  LOCATESYSTEM 2.3-57
  LOCKFILE 2.3-58
  MONITORNET 2.3-61
  MONITORNEW 2.3-62
  NEXTFILENAME 2.3-63
  OPEN 2.3-65
  POSITION 2.3-73
  PURGE 2.3-75
INDEX

READ 2.3-76
READUPDATE 2.3-79
RECEIVEINFO 2.3-82
REFRESH 2.3-85
REMOTEPROCESSORSTATUS 2.3-86
RENAME 2.3-88
REPLY 2.3-89
REPOSITION 2.3-91
SAVEPOSITION 2.3-92
security checking (disc files) 2.3-6
SETMODE 2.3-93
SETMODENOWAIT 2.3-95
UNLOCKFILE 2.3-107
WRITE 2.3-108
WRITEREAD 2.3-110
WRITEUPDATE 2.3-112
FILEERROR procedure 2.3-36
FILEINFO procedure 2.3-39, 2.4-1
Files 2.1-1
  disc files 2.1-1
  interprocess communication 2.1-4
  non-disc devices 2.1-3
  operator console 2.1-7
FIXSTRING procedure 4-4
  considerations 4-8
  implementing an FC command 4-8
  subcommands 4-5
Floating priorities 3.1-7
FNAMECOLLAPSE procedure 2.3-43
FNAMECOMPARE procedure 2.3-45
FNAMEEXPAND procedure 2.3-48
  expansion summary 2.3-46
  network file names 2.3-49
Format, formatter 10-14
FORMATCONVERT procedure 10-2
FORMATDATA procedure 10-5
Formatter 10-1
  format characteristics 10-14
  "A" edit descriptor 10-26
  "D" edit descriptor 10-28
  "E" edit descriptor 10-28
  "F" edit descriptor 10-31
  "G" edit descriptor 10-32
  "I" edit descriptor 10-34
  "L" edit descriptor 10-35
  "M" edit descriptor 10-37
  blank descriptors 10-24
  buffer control descriptors 10-24
  decorations 10-44
  edit descriptors 10-17
  field blanking modifiers 10-40
  fill character modifier 10-40
  justification modifiers 10-41
  literal descriptors 10-21
INDEX

modifiers 10-40
non-repeatable edit descriptors 10-20
optional plus descriptors 10-23
overflow character modifier 10-41
repeatable edit descriptors 10-26
scale factor descriptor 10-22
symbol substitution modifier 10-42
tabulation descriptors 10-20
FORMATCONVERT procedure 10-2
FORMATDATA procedure 10-5
introduction 10-1
list-directed formatting 10-48
input 10-48
output 10-49

GETCRTPID procedure 3.2-12
GETDEVNAME procedure 2.3-52
GETPOOL procedure 8.1-8
GETPPDENTRY procedure 3.2-13
GETREMOCKETRTPID procedure 3.2-15
GETSYNCINFO procedure 5.2-18
GETSYSTEMNAME procedure 2.3-54
GIVE command (FUP) 7.1-10, 7.1-13
GIVE~BREAK procedure 9-14
Group manager 7.1-2

HEAPSORT procedure 4-11
Home terminal 1-6, 3.1-18

I/O structure
  hardware 2.1-16
  software 2.1-18
INFO command (FUP) 7.1-10, 7.1-13
INITIALIZER procedure 4-13, 9-46
  backup process 4-16
  primary process 4-15
Interface with INITIALIZER and ASSIGNS 9-46
  considerations 9-49
  INITIALIZER-related defines 9-46
  setting file characteristics 9-49
  usage example 9-50
Interprocess communication 2.1-4, 2.9-1
  $RECEIVE file 2.9-7
    communication type 2.9-9
    no-wait i/o 2.9-7
    system message transfer 2.9-8
    applicable procedures 2.9-4
    characteristics 2.9-1
    communication synchronization 2.9-6
    error recovery 2.9-31
    example 2.9-19
    one-way communication 2.9-5
    system messages 2.9-25
    two-way communication 2.9-5

Index-8
 Introduction to GUARDIAN  1-1

LASTADDR procedure  4-17
LASTRECEIVE procedure  2.3-55
LICENSE command (FUP)  7.1-10, 7.1-13
Licensing  7.1-9
Line printers  2.6-1
    accessing  2.6-2
    applicable procedures  2.6-2
    characteristics  2.6-1
    CONTROL operations  2.6-17
    CONTROLBUF operations  2.6-18
    error recovery  2.6-15
    forms control  2.6-3
    model 5508 programming considerations  2.6-5
    model 5520 condensed print  2.6-11
    model 5520 expanded print  2.6-11
    model 5520 programming considerations  2.6-6
    path error recovery  2.6-16
    SETMODE operations  2.6-18
    using model 5508 over phone lines  2.6-6/14
    using model 5520 over phone lines  2.6-14
Link control blocks (LCB's)  2.11-1
LOCATESYSTEM procedure  2.3-57
LOCKDATA procedure  8.2-2
LOCKFILE procedure  2.3-58
Locking disc files  2.1-12
LOCKMEMORY procedure  8.2-5
Logging on  7.1-4
Logical device numbers  2.2-3
Logical Device Table  2.1-21
LOGOFF command  7.1-10
LOGON command  7.1-10
LOOKUPPROCESSNAME procedure  3.2-16
    network use  3.2-17
Loop detection, in a process  3.1-7

Magnetic tapes  2.7-1
    accessing  2.7-3
    applicable procedures  2.7-3
    BOT marker  2.7-5
    characteristics  2.7-1
    concepts  2.7-4
    CONTROL operations  2.7-17
    EOT marker  2.7-5
    error recovery  2.7-14
    files  2.7-5
    records  2.7-6
    seven-track tape conversion  2.7-18
    short write mode  2.7-24
Memory management procedures  8.1-1
    advanced  8.2-1
    ALLOCATESEGMENT  8.1-4
    DEALLOCATESEGMENT  8.1-6
INDEX

DEFINEPOOL 8.1-7
GETPOOL 8.1-8
LOCKDATA 8.2-2
LOCKMEMORY 8.2-5
PUTPOOL 8.1-9
UNLOCKMEMORY 8.2-8
USESEGMENT 8.1-10
Memory segments, extended 8.1-1
space management within 8.1-3
Message Format
operator console 2.10-3
Mirror volumes 2.1-34
Modems 2.5-23
accessing 5508 line printers over 2.6-6
Modifiers, formatter
BN, blank null 10-40
BZ, blank zero 10-40
FL, fill character 10-40
LJ, left justification 10-42
OC, overflow character 10-41
RJ, right justification 10-42
SS, symbol substitution 10-42
MOM procedure 3.2-18
MONITORCPUS procedure 5.2-19
MONITORNET procedure 2.3-61
MONITORNEW procedure 2.3-62
MYPID procedure 3.2-20
MYSYSTEMNUMBER procedure 3.2-21
MYTERM procedure 3.2-22
Named process pairs 5.3-1
process startup for 5.3-1
Named processes 3.1-12
ancestor process 3.1-13
backup process 3.1-12
operation of the PPD 3.1-12
primary process 3.1-12
Networks, EXPAND
file names 2.2-7, 11-9
process access 7.1-17
programmatically logging on 7.1-17
security 7.1-14
NEWPROCESS procedure 3.2-23
errors 3.2-32
NEWPROCESNOWAIT procedure 3.2-28
errors 3.2-32
NEXTFILENAME procedure 2.3-63
No-wait CHECKOPEN feature 5.2-10
No-wait i/o 2.1-13
with sequential i/o procedures 9-63
No-wait OPEN feature 2.3-69
Non-named process pairs 5.3-1
process startup for 5.3-9
Non-named processes 3.1-8

Index-10
Non-retryable operations 2.1-28
NonStop operation 1-1
NonStop programs 1-8, 2.9-9, 5.1-1, 5.3-1
even 12.1-1
NOERROR procedure 9-56
error handling 9-58
NUMIN procedure 4-18
NUMOUT procedure 4-21
OPEN procedure 2.3-65
Opening a file 2.1-21
in a NonStop program 5.3-13
OPENFILE procedure 9-15
example 9-20
flags 9-16
Operator console 2.1-7
Operator Console 2.10-1
Operator console 2.10-1
applicable procedures 2.10-2
characteristics 2.10-1
error recovery 2.10-3
logging to an application process 2.10-3
message format 2.10-3
writing a message 2.10-2
Operator, system 7.1-2
Paired opening of files 2.9-10, 5.1-3, 5.3-13
PARAM command 11-22
param message 11-23
Passing parameter information 11-11
PASSWORD command 7.1-10, 7.1-12
Passwords 7.1-5
Path error recovery 2.1-28, 2.4-29
for card readers 2.8-7
for line printers 2.6-16
for magnetic tapes 2.7-16
for operator console 2.10-3
for process files 2.9-31
for terminals 2.5-36
POSITION procedure 2.3-73
Primary process 1-8, 2.9-10, 3.1-12, 5.1-1, 5.3-1
Printers 2.6-1
accessing 2.6-2
applicable procedures 2.6-2
characteristics 2.6-1
CONTROL operations 2.6-17
CONTROLBUF operations 2.6-18
derror recovery 2.6-15
forms control 2.6-3
model 5508 programming considerations 2.6-5
model 5520 programming considerations 2.6-6
path error recovery 2.6-16
SEMODE operations 2.6-18
using model 5520 over phone lines 2.6-6, 2.6-11
Priorities, execution 3.1-6, 3.4-1  
floating 3.1-7  
PRIORITY procedure 3.2-34  
Procedures  
checkpointing 5.2-1  
file system 2.11-1, 2.3-1  
formatter 10-1  
memory management 8.1-1, 8.2-1  
process control 3.2-1  
security system 7.2-1  
sequential i/o 9-1  
syntax summary A-1  
trap handling 6-4  
utility 4-1  
Process 3.1-1  
creation 1-5, 3.1-5, 3.3-1  
deletion 1-6, 3.1-7  
exectution 3.1-6  
startup for named process pairs 5.3-1  
startup for non-named process pairs 5.3-9  
states 3.1-5  
structure 1-7  
Process accessor ID 7.1-5  
Process control 1-5, 3.1-1  
Process control procedures  
ABEND 3.2-3  
ACTIVATEPROCESS 3.2-4  
ALTERPRIORITY 3.2-5  
CANCELTIMEOUT 3.2-6  
CONVERTPROCESSNAME 3.2-7  
CREATEGROUPNAME 3.2-8  
CREATEPROCESSNAME 3.2-10  
DELAY 3.2-11  
GETCRTPID 3.2-12  
GETPPDENTRY 3.2-13  
GETREMOTECRTPID 3.2-15  
LOOKUPPROCESSNAME 3.2-16  
MOM 3.2-18  
MYPID 3.2-20  
MYSYSTEMNUMBER 3.2-21  
MYTERM 3.2-22  
NEWPROCESS 3.2-23  
NEWPROCESSNOWAIT 3.2-28  
PRIORITY 3.2-34  
PROCESSINFO 3.2-35  
PROGRAMFILENAME 3.2-38  
SETLOOPTIMER 3.2-39  
SETMYTERM 3.2-42  
SETSTOP 3.2-43  
SIGNALTIMEOUT 3.2-44  
STEPMOM 3.2-46  
STOP 3.2-48  
SUSPENDPROCESS 3.2-49  
Process files 2.9-9
INDEX

Process ID 1-6, 2.2-4, 3.1-8
   obtaining a 2.2-5
Process name form of process ID
   local 1-6, 2.2-4, 3.1-8
   network 1-6, 2.2-4, 3.1-9
Process pairs 1-8, 3.1-10
Process-Pair Directory (PPD) 1-6, 3.1-12
PROCESSACCESSID procedure 7.2-3
PROCESSINFO procedure 3.2-35
Processor failure 1-8, 3.1-14, 5.2-1, 5.3-22
PROCESSORSTATUS procedure 5.2-21
Program 3.1-1
PROGRAMFILENAME procedure 3.2-38
Programmatically logging on 7.1-17
Pseudo-polling for terminals 2.5-19
   simulation of 2.5-20
PURGE procedure 2.3-75
PUTPOOL procedure 8.1-9

READ procedure 2.3-76
Reading parameter messages 11-26
READUPDATE procedure 2.3-79
Ready list 3.1-6
Ready state, of a process 3.1-6
READFILE procedure 9-21
RECEIVEINFO procedure 2.3-82
REFRESH procedure 2.3-85
Remote passwords 7.1-15
REMOTEPASSWORD command 7.1-10
REMOTEPROCESSORSTATUS procedure 2.3-86
RENAME procedure 2.3-88
REPLY procedure 2.3-89
REPOSITION procedure 2.3-91
Requestor ID 2.1-28
Requestors 1-7, 2.9-19, 12.1-1
Reserved link control blocks 2.11-1
RESERVELCBS procedure 2.11-3
RESETSYNC procedure 5.2-22
Resident buffering 2.11-5
Retryable operations 2.1-28, 2.4-29
REVOKE command (FUP) 7.1-10, 7.1-13
RUN command 11-12

SAVEPOSITION procedure 2.3-92
SECURE command (FUP) 7.1-10, 7.1-13
Security system 1-17, 7.1-1
   accessor ID's 7.1-5
   default security for disc files 7.1-7
   defining users 7.1-3
   disc file security 2.3-69, 7.1-6

Index-13
interface to 7.1-10, 7.2-1
  Command Interpreter 7.1-10
  file system procedures 7.1-11, 7.2-4
  FUP 7.1-10
  process control procedures 7.1-11, 7.2-7
  security system procedures 7.1-11, 7.2-2, 7.2-8
  levels of security 7.1-6
  licensing 7.1-9
  local and remote processes 7.1-7
  logging on 7.1-4
  naming conventions 7.1-4
  passwords 7.1-5
  procedures 7.2-1
    CREATORACCESSID 7.2-2
    PROCESSACCESSID 7.2-3
    SETMODE 2.3-93, 7.2-4
    SETMODENOWAIT 2.3-95, 7.2-4
    SETSTOP 3.2-43, 7.2-7
    USERIDTOUSERNAME 7.2-8
    USERNAMETOUUSERID 7.2-9
    VERIFYUSER 7.2-10
  system users 7.1-2
Segments, extended memory 8.1-1
  space management within 8.1-3
Sequential i/o procedures 9-1
  CHECKBREAK 9-4
  CHECKFILE 9-5
  CLOSEFILE 9-12
  errors 9-38
  FCB structure 9-41, E-1
  GIVEBREAK 9-14
  initializing the file FCB 9-42
  interface with INITIALIZER and ASSIGNs 9-46
  considerations 9-49
    INITIALIZER-related defines 9-46
  summary 9-53
  NOERROR 9-56
  OPENFILE 9-15
  READFILE 9-21
  SETFILE 9-23
  TAKEBREAK 9-33
  usage example
    with INITIALIZER and ASSIGN messages 9-50
    without INITIALIZER procedure 9-54
  WAITFILE 9-34
  WRITEFILE 9-36
Servers 1-7, 2.9-19, 12.1-1
SETLOOPTIMER procedure 3.2-39
SETMODE functions table 2.3-97
SETMODE procedure 2.3-93
  functions 2.3-97
  security aspects 7.2-4

Index-14
SETMODENOWAIT procedure 2.3-95
  functions 2.3-97
  security aspects 7.2-4
SETMYTERM procedure 3.2-42
SETSTOP procedure 3.2-43
  security aspects 7.2-7
SETSINCINFO procedure 5.2-23
SET^FILE procedure 9-23
  operations 9-24
Seven-track tape conversion modes
  ASCIIBC 2.7-18
  BINARY1TO1 2.7-23
  BINARY2TO3 2.7-22
  BINARY3TO4 2.7-21
  selecting a conversion mode 2.7-23
SHIFTSTRING procedure 4-23
Short write mode, for magnetic tapes 2.7-24
SIGNALTIMEOUT procedure 3.2-44
Startup message 11-14
STEMPOM procedure 3.2-46
STOP procedure 3.2-48
Super ID 7.1-2
Suspended state, of a process 3.1-6
SUSPENDPROCESS procedure 3.2-49
Sync block 5.1-5
Sync ID 2.1-28, 2.9-12
Syntax summary, of procedures A-1
System messages 1-13, 2.9-25, 3.1-14, 5.3-22, C-1
System name 2.2-8, 2.3-54
System number 2.2-7
System operator 7.1-2

TAKE^BREAK procedure 9-33
Tapes 2.7-1
  accessing 2.7-3
  applicable procedures 2.7-3
  BOT marker 2.7-5
  characteristics 2.7-1
  concepts 2.7-4
  CONTROLBUF operations 2.7-18
  BOT marker 2.7-5
  error recovery 2.7-14
  files 2.7-5
  records 2.7-6
  seven-track tape conversion 2.7-18
  short write mode 2.7-24
Terminals 2.5-1
  accessing 2.5-4
  applicable procedures 2.5-3
  characteristics 2.5-1
  checksum processing 2.5-22
  configuration parameters 2.5-36
  CONTROL operations 2.5-37
INDEX

conversational mode 2.5-8
  forms control 2.5-14
  interrupt characters 2.5-10
  line termination character 2.5-9
echo 2.5-22
error recovery 2.5-34
modems 2.5-23
page mode 2.5-16
  interrupt characters 2.5-16
  page termination character 2.5-16
  pseudo-pollled terminals 2.5-19
  simulation of pseudo-polling 2.5-20
SETMODE operations 2.5-37
timeouts 2.5-23
transfer modes 2.5-6
transparency mode 2.5-22
TIME procedure 4-24
Timeout
  elapsed time 3.1-19
  for no-wait i/o 2.1-14
  for process suspension 3.1-7
  for terminals 2.5-23
  process execution time 3.1-7
Timestamp form of process ID 1-6, 2.2-4, 3.1-8
TIMESTAMP procedure 4-25
TOSVERSION procedure 4-26
Trap handling 1-16, 6-3
  ARMTRAP procedure 6-4
Traps 1-16, 6-1
Tri-Density Tape Subsystem
  Microcode 2.7-11
  Model 5106 2.7-11

UNLOCKFILE procedure 2.3-107
UNLOCKMEMORY procedure 8.2-8
USERIDTOUSERNAME procedure 7.2-8
USERNAMETOUERID procedure 7.2-9
USERS command 7.1-10, 7.1-12
USESEGMENT procedure 8.1-10
Utility procedures 1-13, 4-1
  CONTIME 4-2
  DEBUG 4-3
  FIXSTRING 4-4
  HEAPSORT 4-11
  INITIALIZER 4-13
  LASTADDR 4-17
  NUMIN 4-18
  NUMOUT 4-21
  SHIFTSTRING 4-23
  TIME 4-24
  TIMESTAMP 4-25
  TOSVERSION 4-26

Index-16
VERIFYUSER procedure 7.2-10
VOLUME command 7.1-10, 7.1-12

Wait and no-wait i/o 2.1-13
Waiting state, of a process 3.1-6
WAIT^FILE procedure 9-34
Wakeup message 11-28
WRITE procedure 2.3-108
WRITERead procedure 2.3-110
WRITEUPDATE procedure 2.3-112
WRITE^FILE procedure 9-36

$0 2.10-1, 2.2-6
$CMON
   logon message 11-31
   process creation message 11-32
$RECEIVE file 2.9-7
   communication type 2.9-9
   data transfer protocol 9-60
   handling by sequential i/o 9-60
   no-wait i/o 2.9-7
   system message transfer 2.9-8
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):

FROM:

Name ________________________________

Company ________________________________

Address ________________________________

City/State ___________________________ Zip ___________

A written response is requested. yes no ?