FastSort Manual

Abstract

This manual describes FastSort, the HP sort-merge utility for HP NonStop™ systems. The *FastSort Manual* is intended for users who sort interactively, programmatically, and from HP NonStop SQL/MP.

Product Version

FastSort D32

Supported Release Version Updates (RVUs)

This publication supports G06.21 and all subsequent G-series RVUs until otherwise indicated by its replacement publication. To use increased Enscribe limits, the minimum RVUs are H06.28 and J06.17 with specific SPRs. For a list of the required SPRs, see SPR Requirements for Increased Enscribe Limits for the H06.28/J06.17 Release.

Part Number	Published
429834-003	April 2014

Document History

Part Number	Product Version	Published
060035	FastSort C30	July 1992
118812	FastSort D40	December 1995
124077	FastSort D32	February 1996
429834-001	FastSort D32	July 2001
429834-002	FastSort D32	September 2003
429834-003	FastSort D32	April 2014

Legal Notices

Printed in the US

FastSort Manual

Glossary | Index | Examples | Figures | Tables

Examples

Figures

Tables

What’s New in This Manual

Manual Information

Abstract

This manual describes FastSort, the HP sort-merge utility for HP NonStop™ systems. The *FastSort Manual* is intended for users who sort interactively, programmatically, and from HP NonStop SQL/MP.

Product Version

FastSort D32

Supported Release Version Updates (RVUs)

This publication supports G06.21 and all subsequent G-series RVUs until otherwise indicated by its replacement publication. To use increased Enscribe limits, the minimum RVUs are H06.28 and J06.17 with specific SPRs. For a list of the required SPRs, see SPR Requirements for Increased Enscribe Limits for the H06.28/J06.17 Release.

Part Number	Published
429834-003	April 2014

Document History

Part Number	Product Version	Published
060035	FastSort C30	July 1992
118812	FastSort D40	December 1995
124077	FastSort D32	February 1996
429834-001	FastSort D32	July 2001
429834-002	FastSort D32	September 2003
429834-003	FastSort D32	April 2014

New and Changed Information

Changes to the 429834-003 manual:

- Updated RECORD length on page 16.
- Updated Output File Types on page 30.
- Updated `buffer2` input and `buffer-length` input on page 5-4.

- Added a section Using 32 KB Buffers on page 5-5.
- Updated block size in the section Improving Performance With Record Blocking and Nowait I/O on page 5-6.
- Updated Using Supported File Types on page C-1.
- Updated Key-Sequenced Files on page C-3.

Previous publication was updated to reflect new product names:

- Since product names are changing over time, this publication might contain both HP and Compaq product names.
- Product names in graphic representations are consistent with the current product interface.
- The technical content of this publication has not been updated and reflects the state of the product at the G06.21 release version update (RVU).

About This Manual

This manual is a combination user's guide and reference manual that describes FastSort, the sort-merge utility for NonStop systems.

SPR Requirements for Increased Enscribe Limits for the H06.28/J06.17 Release

As of H06.28 and J06.17 RVUs, format 2 legacy key-sequenced 2 (LKS2) files with increased limits, format 2 standard queue files with increased limits, and enhanced key-sequenced (EKS) files with increased limits are introduced. EKS files with increased limits support 17 to 128 partitions along with larger record, block, and key sizes. LKS2 files with increased limits and format 2 standard queue files with increased limits support larger record, block, and key sizes. When a distinction is not required between these file types, key-sequenced files with increased limits are used as a collective term. To achieve these increased limits with H06.18 and J06.17 RVUs, the following SPRs are required: (These SPR requirements could change or be eliminated with subsequent RVUs.)

Products	**J-Series SPR**	**H-Series SPR**
Backup/Restore NSK	T9074H01^AGJ	T9074H01^AGJ
DP2	T9053J02^AZZ	T9053H02^AZN
File System	T9055J05^AJQ	T9055 H14^AJP
FUP	T6553H02^ADH	T6553H02^ADH
NS TM/MP TMFDR	T8695J01^ALP	T8695H01^ALO
SMF	T8472H01^ADO	T8472H01^ADO
	T8471H01^ADO	T8471H01^ADO
	T8470H01^ADO	T8470H01^ADO
	T8469H01^ADO	T8469H01^ADO
	T8466H01^ADO	T8466H01^ADO
	T8465H01^ADO	T8465H01^ADO
	T8468H01^ABY	T8468H01^ABY
SQL/MP	T9191J01^ACY	T9191H01^ACX
	T9195J01^AES	T9195H01^AER
	T9197J01^AEA	T9197H01^ADZ
TCP/IP FTP	T9552H02^AET	T9552H02^AET
TNS/E COBOL Runtime Library	T0357H01^AAO	T0357H01^AAO

Audience

This manual is intended for all FastSort users, including:

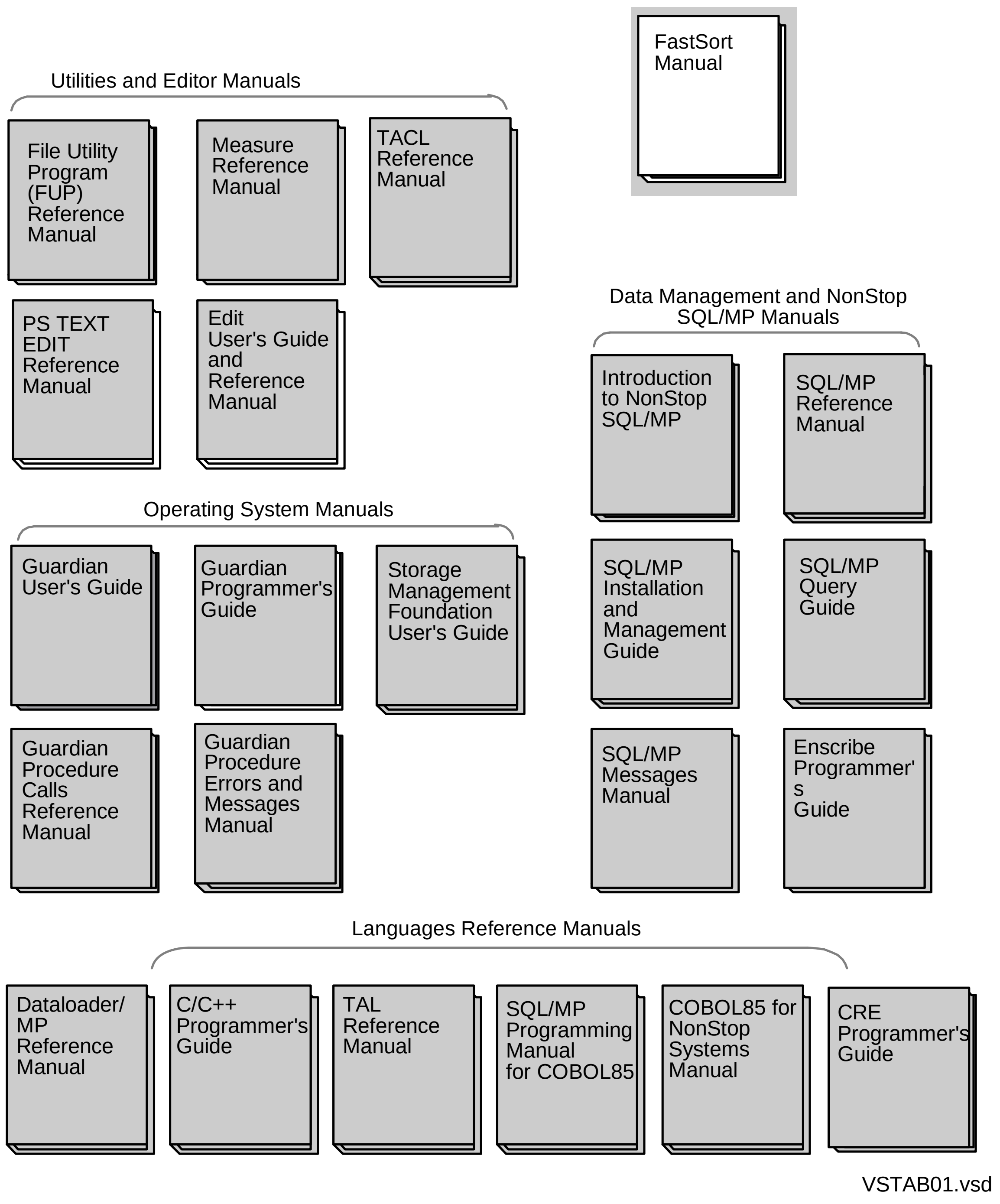
- Users who issue FastSort interactive commands from a terminal or through a command file
- Programmers who call FastSort system procedures from an application program (including COBOL85 programmers who use the SORT and MERGE statements)
- NonStop SQL/MP users, if SQL/MP is installed on your system and you initiate queries that sort entries or load data

A reader of this manual should be familiar with the NonStop Kernel operating system, File Utility Program (FUP), Enscribe database manager, and SQL/MP (if used).

Related Manuals

While using the *FastSort Manual*, you might need to refer to one or more of the manuals shown in Figure i on page xi.

Figure i. Related Manuals


FastSort Manual
Utilities and Editor Manuals
File Utility Program (FUP) Reference Manual
Measure Reference Manual
TACL Reference Manual
PS TEXT EDIT Reference Manual
Edit User's Guide and Reference Manual
Data Management and NonStop SQL/MP Manuals
Introduction to NonStop SQL/MP
SQL/MP Reference Manual
Operating System Manuals
Guardian User's Guide
Guardian Programmer's Guide
Storage Management Foundation User's Guide
SQL/MP Installation and Management Guide
SQL/MP Query Guide
Guardian Procedure Calls Reference Manual
Guardian Procedure Errors and Messages Manual
SQL/MP Messages Manual
Enscribe Programmer's Guide
Languages Reference Manuals
Dataloader/MP Reference Manual
C/C++ Programmer's Guide
TAL Reference Manual
SQL/MP Programming Manual for COBOL85
COBOL85 for NonStop Systems Manual
CRE Programmer's Guide
VSTAB01.vsd

Notation Conventions

Hypertext Links

Blue underline is used to indicate a hypertext link within text. By clicking a passage of text with a blue underline, you are taken to the location described. For example:

This requirement is described under Backup DAM Volumes and Physical Disk Drives on page 3-2.

General Syntax Notation

This list summarizes the notation conventions for syntax presentation in this manual.

UPPERCASE LETTERS. Uppercase letters indicate keywords and reserved words. Type these items exactly as shown. Items not enclosed in brackets are required. For example:

```
MAXATTACH
```

lowercase italic letters. Lowercase italic letters indicate variable items that you supply. Items not enclosed in brackets are required. For example:

```
file-name
```

computer type. `Computer type` letters within text indicate C and Open System Services (OSS) keywords and reserved words. Type these items exactly as shown. Items not enclosed in brackets are required. For example:

```
myfile.c
```

italic computer type. *`Italic computer type`* letters within text indicate C and Open System Services (OSS) variable items that you supply. Items not enclosed in brackets are required. For example:

```
pathname
```

[ ] Brackets. Brackets enclose optional syntax items. For example:

```
TERM [\system-name.]$terminal-name

INT[ERRUPTS]
```

A group of items enclosed in brackets is a list from which you can choose one item or none. The items in the list can be arranged either vertically, with aligned brackets on each side of the list, or horizontally, enclosed in a pair of brackets and separated by vertical lines. For example:

```
FC [ num  ]
   [ -num ]
   [ text ]

K [ X | D ] address
```

{ } Braces. A group of items enclosed in braces is a list from which you are required to choose one item. The items in the list can be arranged either vertically, with aligned braces on each side of the list, or horizontally, enclosed in a pair of braces and separated by vertical lines. For example:

```
LISTOPENS PROCESS { $appl-mgr-name }
                  { $process-name  }
```

```
ALLOWSU { ON | OFF }
```

| Vertical Line. A vertical line separates alternatives in a horizontal list that is enclosed in brackets or braces. For example:

```
INSPECT { OFF | ON | SAVEABEND }
```

… Ellipsis. An ellipsis immediately following a pair of brackets or braces indicates that you can repeat the enclosed sequence of syntax items any number of times. For example:

```
M address [ , new-value ]…
```

```
[ - ] {0|1|2|3|4|5|6|7|8|9}…
```

An ellipsis immediately following a single syntax item indicates that you can repeat that syntax item any number of times. For example:

```
"s-char…"
```

Punctuation. Parentheses, commas, semicolons, and other symbols not previously described must be typed as shown. For example:

```
error := NEXTFILENAME ( file-name ) ;
```

```
LISTOPENS SU $process-name.#su-name
```

Quotation marks around a symbol such as a bracket or brace indicate the symbol is a required character that you must type as shown. For example:

```
"[" repetition-constant-list "]"
```

Item Spacing. Spaces shown between items are required unless one of the items is a punctuation symbol such as a parenthesis or a comma. For example:

```
CALL STEPMOM ( process-id ) ;
```

If there is no space between two items, spaces are not permitted. In this example, no spaces are permitted between the period and any other items:

```
$process-name.#su-name
```

Line Spacing. If the syntax of a command is too long to fit on a single line, each continuation line is indented three spaces and is separated from the preceding line by a blank line. This spacing distinguishes items in a continuation line from items in a vertical list of selections. For example:

```
ALTER [ / OUT file-spec / ] LINE

   [ , attribute-spec ]…
```

!i and !o. In procedure calls, the !i notation follows an input parameter (one that passes data to the called procedure); the !o notation follows an output parameter (one that returns data to the calling program). For example:

```
CALL CHECKRESIZESEGMENT (  segment-id                    !i
                         , error        ) ;              !o
```

!i,o. In procedure calls, the !i,o notation follows an input/output parameter (one that both passes data to the called procedure and returns data to the calling program). For example:

```
error := COMPRESSEDIT ( filenum ) ;                      !i,o
```

!i:i. In procedure calls, the !i:i notation follows an input string parameter that has a corresponding parameter specifying the length of the string in bytes. For example:

```
error := FILENAME_COMPARE_ (  filename1:length           !i:i
                            , filename2:length ) ;       !i:i
```

!o:i. In procedure calls, the !o:i notation follows an output buffer parameter that has a corresponding input parameter specifying the maximum length of the output buffer in bytes. For example:

```
error := FILE_GETINFO_ (  filenum                        !i
                        , [ filename:maxlen ] ) ;        !o:i
```

Notation for Messages

This list summarizes the notation conventions for the presentation of displayed messages in this manual.

Bold Text. Bold text in an example indicates user input typed at the terminal. For example:

```
ENTER RUN CODE

?123

CODE RECEIVED:      123.00
```

The user must press the Return key after typing the input.

Nonitalic text. Nonitalic letters, numbers, and punctuation indicate text that is displayed or returned exactly as shown. For example:

```
Backup Up.
```

lowercase italic letters. Lowercase italic letters indicate variable items whose values are displayed or returned. For example:

p-register

process-name

[ ] Brackets. Brackets enclose items that are sometimes, but not always, displayed. For example:

```
Event number = number [ Subject = first-subject-value ]
```

A group of items enclosed in brackets is a list of all possible items that can be displayed, of which one or none might actually be displayed. The items in the list can be arranged either vertically, with aligned brackets on each side of the list, or horizontally, enclosed in a pair of brackets and separated by vertical lines. For example:

```
proc-name trapped [ in SQL | in SQL file system ]
```

{ } Braces. A group of items enclosed in braces is a list of all possible items that can be displayed, of which one is actually displayed. The items in the list can be arranged either vertically, with aligned braces on each side of the list, or horizontally, enclosed in a pair of braces and separated by vertical lines. For example:

```
obj-type obj-name state changed to state, caused by
{ Object | Operator | Service }
```

```
process-name State changed from old-objstate to objstate
{ Operator Request. }
{ Unknown.          }
```

| Vertical Line. A vertical line separates alternatives in a horizontal list that is enclosed in brackets or braces. For example:

```
Transfer status: { OK | Failed }
```

% Percent Sign. A percent sign precedes a number that is not in decimal notation. The % notation precedes an octal number. The %B notation precedes a binary number. The %H notation precedes a hexadecimal number. For example:

```
%005400
```

```
%B101111
```

```
%H2F
```

```
P=%p-register E=%e-register
```

Change Bar Notation

Change bars are used to indicate substantive differences between this manual and its preceding version. Change bars are vertical rules placed in the right margin of changed portions of text, figures, tables, examples, and so on. Change bars highlight new or revised information. For example:

> The message types specified in the REPORT clause are different in the COBOL85 environment and the Common Run-Time Environment (CRE).
>
> The CRE has many new message types and some new message type codes for old message types. In the CRE, the message type SYSTEM includes all messages except LOGICAL-CLOSE and LOGICAL-OPEN.

HP Encourages Your Comments

HP encourages your comments concerning this document. We are committed to providing documentation that meets your needs. Send any errors found, suggestions for improvement, or compliments to docsfeedback@hp.com.

Include the document title, part number, and any comment, error found, or suggestion for improvement you have concerning this document.

1 Introduction to FastSort

FastSort is the sort-merge tool for HP NonStop systems. FastSort can sort or merge records in one of two modes:

- A serial operation uses one SORTPROG process in one processor with up to 32 individual scratch files or a single partitioned scratch file.
- A parallel operation distributes the sort or merge workload across multiple SORTPROG processes, processors, and scratch files.

Sort and Merge Operations

A sort operation arranges and combines one or more sets of input records into a single set of output records. During a sort operation, FastSort arranges the records in either ascending or descending order, or in a combination of both based on a sequence of key-field values.

A merge operation combines two or more sets of sorted input records into a single set of output records. The records for merging are already sorted in an ascending or descending sequence of key-field values.

FastSort accepts records to sort or merge from these sources:

- 1 to 32 disk files
- A terminal
- An application process
- Tape files

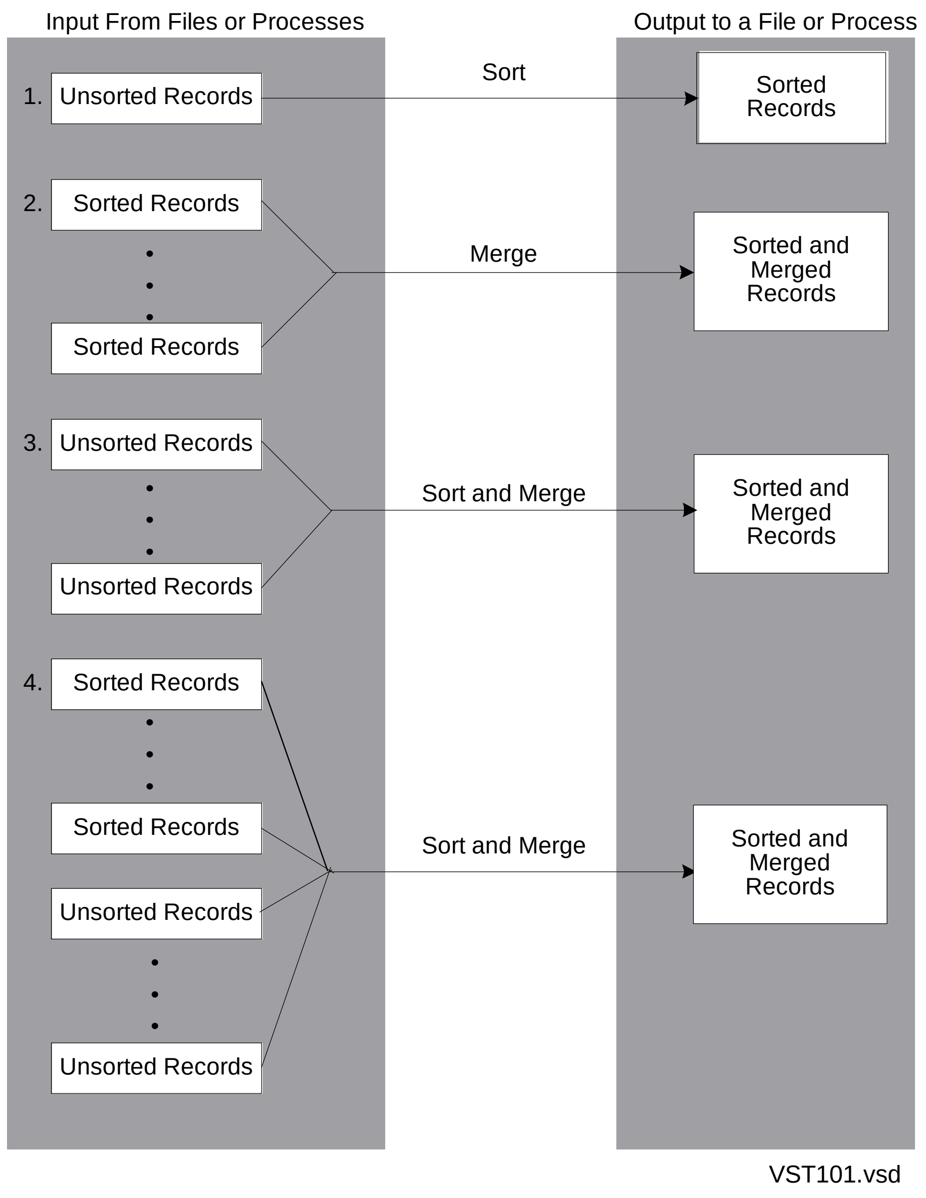
You use FastSort interactive commands or system procedures to define the sort or merge operation. In each sort or merge run, FastSort performs one of the following operations:

- Sorts one set of input records and produces one set of output records
- Merges two or more sets of sorted records into one set of output records
- Sorts two or more sets of input records and merges them into one set of output records
- Sorts one or more sets of input records and merges them with one or more sets of sorted input records into one set of output records

Figure 1-1 on page 1-2 shows these four FastSort operations.

After sorting and merging all the input records, FastSort returns the output records to your application process or writes them to a file or a terminal. You can also have FastSort return output records that are sequence numbers or key-field values, or both, instead of entire records.

Figure 1-1. FastSort Operations



FastSort Features

FastSort has these major features:

- Key fields
  - Accepts up to 63 alphanumeric or numeric key fields for sorting
  - Recognizes contiguous, noncontiguous, and overlapping key fields

- Sequence options
    - Can use an alternate collating sequence for alphanumeric characters
    - Can remove records with duplicate key values
- Input and output file options
    - Accepts up to 32 input Guardian files (except blocked tape files, SQL objects, or processes)
    - Can merge up to 32 input streams from an application process
    - Supports application process input and output, with optional extended addressing for sending and receiving records
    - Supports partitioned input and output files, including distributed Enscribe databases (but not NonStop SQL/MP objects)
    - Produces relative, entry-sequenced, key-sequenced, or unstructured output files
    - Supports both Format 1 and Format 2 files
- Efficiency
    - Uses an optimized key-comparison algorithm and optimized procedures for reading and building structured files, and double buffering for reading from input files
    - Uses extended memory to minimize the number of merge passes
    - Automatically selects extended memory size
    - Uses record blocking to minimize the number of interprocess messages required for transferring records to or from an application process
    - Supports parallel sorting to minimize sort time for files larger than 1 megabyte
    - Sorts files smaller than 100 kilobytes in memory to eliminate the need for scratch file input and output
    - Provides record blocking and faster sorting for COBOL85, the File Utility Program (FUP), and the CROSSREF program
    - Recognizes SORT and SUBSORT DEFINEs that allow you to control most factors of a FastSort process from outside your sort program.

FastSort Components

FastSort has these major components:

- Interactive FastSort
- Programmatic FastSort

- SORTPROG process (the actual sort-merge process)
- Record generator (RECGEN) process, if SQL/MP is installed on your system

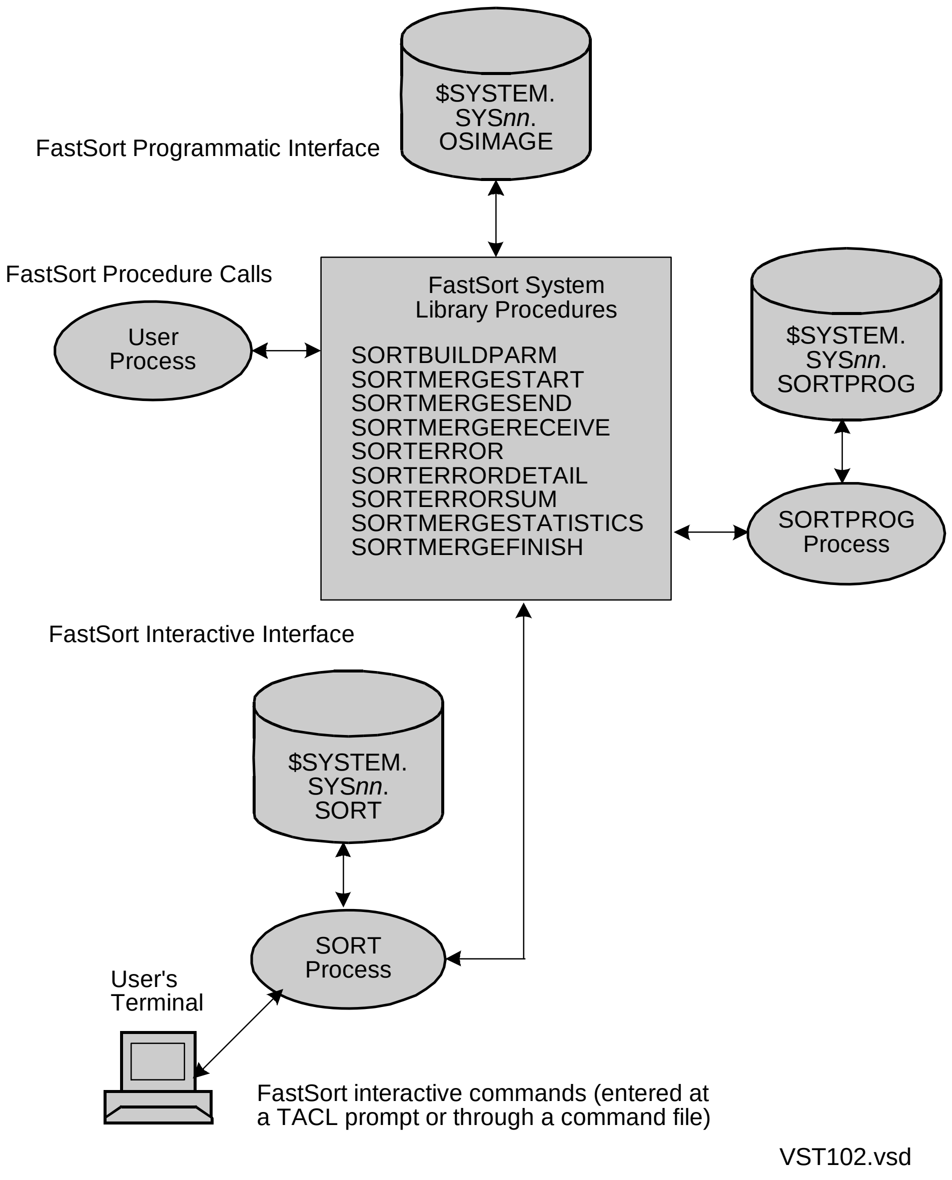
When you want to sort or merge records interactively or through a command file (IN file), you use interactive FastSort. When you want to sort or merge records in an application, you invoke programmatic FastSort through system procedure calls. Interactive or programmatic FastSort can start a SORTPROG process, which performs the sort or merge operation.

Component	**Program File**	**Description**
SORT	$SYSTEM.SYSnn.SORT *	Interactive FastSort – Conversational interface for FastSort commands
System Procedures	$SYSTEM.SYSnn.OSIMAGE	Programmatic FastSort – Library file for FastSort system procedures
SORTPROG	$SYSTEM.SYSnn.SORTPROG *	A single serial sort-merge process, or parallel sort-merge processes including: A distributor-collector process 2 to 8 subsort processes
RECGEN	$SYSTEM.SYSnn.RECGEN *	Record generator (RECGEN) process for the parallel creation and loading of partitioned indexes (if SQL/MP is installed on your system)

In $SYSTEM.SYSnn notation, nn is a two-digit number assigned by HP.
* These program files could also exist in $SYSTEM.SYSTEM.

Figure 1-2 on page 1-5 shows how the first three FastSort components work together.

Figure 1-2. FastSort Components



Interactive FastSort

You can issue FastSort interactive commands from a terminal or through a command file. These commands use FastSort system procedures to communicate with the SORTPROG process. Table 1-1 on page 1-6 summarizes FastSort interactive commands.

Table 1-1. FastSort Interactive Commands (page 1 of 2)

Command	Description
Record Sequence Specification	
ASCENDING	Describes the location and attributes of one or more key fields that determine an ascending sequence for output records.
COLLATE	Specifies a file that contains an alternate collating sequence for comparing key fields.
COLLATEOUT	Stores an alternate collating sequence table in an unstructured file.
DESCENDING	Describes the location and attributes of one or more key fields that determine a descending sequence for output records.
File Specification	
FROM	Specifies the name of an input file for a sort or merge run and the exclusion mode to use to open the file, the maximum number of records in the file, the maximum length of records in the file, and whether the records in the file are already sorted.
TO	Specifies an output file for a sort run and parameters for the file including the percentage of data and index slack, whether FastSort should purge and recreate an existing output file, and the type of sort run (record, permutation, or key sort).
Command Specification	
CLEAR	Deletes current command parameters for all commands or for a specific command.
FC	Displays the last FastSort command for subsequent editing and re-execution.
HELP	Displays the syntax of a specific command or a list of all FastSort commands with a description of each command.
SAVE	Saves FastSort command parameters from a sort or merge run for reuse in subsequent runs.
SHOW	Displays the command parameters currently in effect and whether they are entered for the next sort or merge run or saved from a previous run.
Parallel Sort and Merge Operations	
CPUS	Specifies processors (or CPUs) in which FastSort can run subsort processes.
NOTCPUS	Specifies processors (or CPUs) in FastSort cannot run subsort processes.
SUBSORT	Specifies parameters for a subsort process for a parallel sort or merge run.

Table 1-1. FastSort Interactive Commands (page 2 of 2)

Command	Description
Process Control	
EXIT	Ends the interactive FastSort session (same as Ctrl-Y).
RUN	Starts a sort or merge run and optionally specifies the SORTPROG process start parameters, the allocation of required disk space, and whether duplicate records should be removed.

Programmatic FastSort

Programmatic FastSort consists of system procedures that are called by user-written applications and the interactive SORT process. The FastSort procedures manage the process creation, control, and communication for the SORTPROG process. You can use procedure calls in an application program to specify the same parameters that you can specify with FastSort interactive commands.

Table 1-2 summarizes the procedures in programmatic FastSort, in the order in which you would normally call them.

Table 1-2. FastSort System Procedures

Procedure Name	Description
SORTBUILDPARM	Specifies parameters for parallel sorting and record blocking.
SORTMERGESTART	Begins the SORTPROG process and passes parameters for a sort or merge run from the calling process to SORTPROG.
SORTMERGESEND	Sends input records from the calling process to the SORTPROG process, one for each call.
SORTMERGERECEIVE	Returns output records from the SORTPROG process to the calling process, one for each call.
SORTERROR	Provides the message text for the last FastSort error code returned by a procedure.
SORTERRORDETAIL	Provides the FastSort error code for the most recent error and if an input file caused the error, identifies the input file.
SORTERRORSUM	Provides information that SORTERROR and SORTERRORDETAIL provide and identifies the cause of the most recent error.
SORTMERGESTATISTICS	Reports information about a sort or merge run and ends the run.
SORTMERGEFINISH	Ends the sort or merge run and stops the SORTPROG process.

SORTPROG Process

The SORTPROG process performs all sort or merge operations. It runs separately from an application process or the interactive SORT process. To configure and start a SORTPROG process, you either:

- Issue FastSort interactive commands
- Call FastSort system procedures

The SORTPROG process does not run as a process pair. If a processor failure occurs when the SORTPROG process is running, you must restart the sort or merge run from the beginning. An application program running as a process pair must also restart an interrupted SORTPROG process from the beginning.

FastSort uses or creates these files:

Input files	Sets of records that you give FastSort to sort or merge through local or remote input disk files, a terminal, or tape files.
Scratch file	A temporary work file. FastSort uses scratch files to store runs of records sorted by each SORTPROG process. For large sort runs that require more than one pass, FastSort creates up to 32 total scratch files.
Swap file	The disk file that FastSort uses for swapping data. Data swapping is the process of copying data between physical memory and disk storage.
Output file	The file FastSort creates after a sort or merge run to receive the sorted or merged records.
List file	The file FastSort creates after a sort or merge run that describes the run.

Input Files

FastSort accepts input from EDIT, key-sequenced, relative, entry-sequenced, and unstructured files. You can sort or merge up to 32 input files in a single run. Each file can contain either fixed-length or variable-length records. For a complete description of file types, see the *Guardian Programmer’s Guide* and the *Enscribe Programmer’s Guide*.

If you have records from blocked tape files to sort or merge, you must deblock the records before SORTPROG can process them. You can do this with the FUP COPY command. For information about how to use FUP, see the *File Utility Program (FUP) Reference Manual*.

Scratch Files

FastSort sorts files smaller than 100 kilobytes in memory. For larger input files, FastSort uses up to 32 scratch files to temporarily store groups of records called runs.

You can create a scratch file before you run FastSort, or you can have SORTPROG create one for you. If you manually create a scratch file, SORTPROG leaves the file intact after the sort or merge run. If SORTPROG creates the scratch file, it is a temporary file and SORTPROG automatically purges it after the sort completes.

Once an initial scratch file exists, FastSort creates additional scratch files as needed. If the initial scratch file becomes full, FastSort automatically creates overflow scratch files until either the sort completes or there are 32 total scratch files. A sort operation requires scratch space equal to all output records from the SORTPROG process plus 6 bytes per record for overhead. For more information about scratch files, see Section 9, Optimizing Sort Performance.

Output Files

FastSort can send output to most types of disk files, except EDIT files or NonStop SQL/MP objects. FastSort can send output to a tape file, but it cannot write records to blocked tape files. After a sort or merge run, you can use FUP to load a blocked tape file. For instructions about using FUP, see the *File Utility Program (FUP) Reference Manual.* You can also load your output into an EDIT file using the EDIT GET command. For instructions on how to use the GET command, see Appendix C, Using Supported File Types.

You can have FastSort compute the size of the output file and then create it, or you can specify an existing output file. For more information about output files, see Appendix C, Using Supported File Types.

Using DEFINEs With FastSort

You can use DEFINEs to configure most aspects of a sort or merge operation. FastSort recognizes class SPOOL and class SORT or SUBSORT DEFINEs. DEFINEs are optional.

Class SPOOL DEFINE

FastSort allows a class SPOOL DEFINE for the list file (that is, the TACL RUN command OUT parameter). For example, the following implicit TACL RUN command specifies a SPOOL DEFINE named =*out_file* for the FastSort list file. The TACL ADD DEFINE command first creates the SPOOL DEFINE and sets the LOC attribute.

```
ADD DEFINE =out_file, CLASS SPOOL, LOC \ny.$s.#sort
...
SORT / OUT =out_file /
...
```

For more information about class SPOOL DEFINEs, see *Guardian User’s Guide*.

Class SORT and SUBSORT DEFINEs

You can use class SORT or SUBSORT DEFINEs to configure a sort or subsort. You specify the DEFINE(s) before running a SORTPROG process, and the information is applied to the sort process when it is run. You can use class SORT or SUBSORT DEFINEs to specify information such as the disk volume for the scratch file, the processors to use, and so on.

For example, you can use class SORT or SUBSORT DEFINEs to specify a scratch volume. In the following example, the SUBSORT DEFINE named =SUBSORTA specifies the $DISK02 disk as the initial scratch volume and the $SPOOL disk for the swap file:

```
SET DEFINE CLASS SUBSORT
SET DEFINE SCRATCH $disk02
SET DEFINE SWAP $spool
...
ADD DEFINE =subsorta
...
```

DEFINE information is valid until modified, deleted, or disabled. For more information about using SORT and SUBSORT DEFINEs with FastSort, See Section 7, Using SORT and SUBSORT DEFINEs.

Products That Use FastSort

These HP products use FastSort to perform sort or merge operations:

Product	FastSort Function
COBOL85	Executes a SORT or MERGE statement
CROSSREF Program	Sorts a cross-reference listing
Enform Database Manager	Sorts records for a report
File Utility Program (FUP)	Loads data into a file
SQL/MP	Sorts entries in a query and load data into a table or an index table
Peripheral Utility Program (PUP)	Sorts entries in the free-space table
HP NonStop Transaction Management Facility (TMF)	Manages audit trail information

If SQL/MP is installed on your system, SQL uses FastSort to sort table rows in certain queries. If you issue a SELECT statement with the DISTINCT, ORDER BY, or GROUP BY clause, SQL starts a SORTPROG process to sort the rows.

When you use the SQLCI CREATE INDEX statement, SQL uses FastSort to load data into the target table. When you use the SQLCI LOAD utility to load data into a key-sequenced table and you do not specify the SORTED option, SQL uses FastSort to sort the data.

When you specify PARALLEL EXECUTION ON for either of these statements, the SQL/MP catalog manager (SQLCAT) process starts a RECGEN process for each partition of the base table and a SORTPROG process for each partition of the index. The RECGEN processes read the rows of the base table. SORTPROG processes sort the generated rows and write them to the partitions of the index.

For more information about the CREATE INDEX or LOAD statements, see Section 8, Sorting From NonStop SQL/MP.

2 Sorting Interactively

You can use SORT, the FastSort interactive process, to sort or merge records without writing an application program. FastSort accepts interactive commands from:

- A TACL process
- A command (IN) file
- Running FastSort

To start an interactive SORT process, enter SORT at a TACL prompt:

```
10> SORT
```

This command executes an implicit TACL RUN command that starts the SORT process. SORT displays the FastSort product banner and a “less than” symbol (<) at your terminal:

```
FastSort - T9620D30 - (31OCT94)
COPYRIGHT TANDEM COMPUTERS INCORPORATED 1991 - 1994
<
 ...
```

```
The “less than” symbol (<) is the FastSort prompt. You
communicate with the SORT process by entering FastSort commands
at this prompt.
```

To stop the SORT process, type EXIT at the SORT prompt:

```
<EXIT
11>
```

You can also press Ctrl-Y at the SORT prompt to stop the SORT process.

Using a Command File

A command file is an EDIT file that interacts with the SORT process. When you execute a command file, you do not need to manually enter commands from your terminal. A FastSort command file must contain FastSort commands. It can also contain input records to sort.

When you start a SORT process, you can specify a command file as the IN file and a list file for the OUT file. A list file receives the output from the sort or merge run. The syntax for specifying the command file, list file, and other options in the SORT command is:

```
SORT [ / IN command-file  [ , OUT list-file ]
                          [ , run-option    ] ... / ]
```

`command-file`

is an EDIT file (file code 101) that contains FastSort interactive commands. A command file can also contain the input records for a sort or merge run.

`list-file`

is a disk file, I/O device, SPOOL DEFINE, or a process that receives the output from the sort or merge run. The output file also includes statistics and any error or warning messages. If *`list-file`* already exists, SORT purges its contents and writes the new output to it. If *`list-file`* does not exist, FastSort creates it as an EDIT file.

`run-option`

is a TACL RUN command option, as described in the *TACL Reference Manual*.

For more information, see Automating FastSort Tasks on page 9-15 for an example on how to use a command file to automate DEFINEs.

Entering Commands and Data in a Command File

When you create an EDIT file to use as a command file, enter only one FastSort command on each line. The RUN command must follow all other FastSort commands for a sort or merge run.

You can also enter input records after the RUN command, one on each line, if you do not specify an input (FROM) file for the run. A command file that contains input records can describe only one sort or merge run.

Entering Comments in a Command File

To include a comment in a FastSort command file, enter an exclamation point (!) at the beginning of the comment line. FastSort recognizes all text to the right of an exclamation point as a comment. A line end or second exclamation point ends the comment. Each comment on a new line must begin with an exclamation point, and a comment cannot continue from line to line.

```
! this is a comment
! this also is a comment !
```

If your command file contains input records, do not mix comments with the input records. Instead, place input records after the RUN command in a command file. When FastSort reads input records, it does not recognize the exclamation point as a comment symbol. Instead, FastSort sorts an exclamation point and any comment text as an input record.

Running With Input From a Command File

In the following example, a command file named COMFILE contains FastSort commands and input records. To execute COMFILE and send the output to the list file named LISTFILE, you would enter:

```
10> SORT / IN COMFILE, OUT LISTFILE, NOWAIT /
```

The SORT process reads the commands and data from COMFILE and initiates a SORTPROG process to sort the data. The SORT process uses the FastSort system procedures described in Section 5, Using FastSort System Procedures to communicate with the SORTPROG process. The NOWAIT parameter is optional.

Listed below are the contents of sample command file COMFILE:

```
! Send sorted records to the file named TOFILE.
! Sort in descending order from column 1 to 10.
TO TOFILE
DESC 1 FOR 10
RUN
apple
orange
lemon
grapefruit
banana
grape
watermelon
```

FastSort creates an output data file named TOFILE and a list file named LISTFILE. Listed below are the contents of sample output file TOFILE, which contains the records sorted in descending order.

```
watermelon
orange
lemon
grapefruit
grape
banana
apple
```

Shown below is a sample LISTFILE, which contains:

- The FastSort banner
- The contents of the command file including comments, commands, and data
- Statistics information for the sort run
- Any errors or warnings that occurred during the run

```
FastSort - T9620D30 - (31OCT94)
COPYRIGHT TANDEM COMPUTERS INCORPORATED 1991 - 1994
    1       ! Send sorted records to the file named TOFILE.
    2       ! Sort in descending order from column 1 to 10.
    3       TO TOFILE
    4       DESC 1 FOR 10
    5       RUN
apple
orange
lemon
grapefruit
banana
grape
watermelon

          7  RECORDS                132  MAX RECORD SIZE
      00:03  ELAPSED TIME           166  BUFFER PAGES
      00:00  I/O WAIT TIME            0  INITIAL RUNS
         17  COMPARES                 0  MERGE ORDER
          0  SCRATCH DISK
          0  SCRATCH SEEKS
Errors detected: 0
Warnings detected: 0
```

Specifying Input Records

Interactive FastSort accepts input records for sorting or merging from:

- A command file
- One or more data files you specify using a FROM command before the RUN command
- The terminal, one record on each line, after the RUN command

Specifying Input Files in the FROM Command

You can use the FROM command to specify input records from existing files. For example, the files INPUT1 and INPUT2 contain records to sort and merge in ascending order. The contents of these files are:

File	Contents
INPUT1	lemon, apple, grapefruit
INPUT2	banana, grape, watermelon, orange

After you invoke interactive FastSort, you can specify these files in FROM commands:

```
13> SORT
<FROM INPUT1
<FROM INPUT2
<ASCENDING 1:10
<RUN
```

FastSort sorts and merges the records in the input files, then displays the sorted and merged records and the statistics for the sort run on your terminal, as shown below:

```
apple
banana
grape
grapefruit
lemon
orange
watermelon
          7  RECORDS               132  MAX RECORD SIZE
      00:04  ELAPSED TIME          166  BUFFER PAGES
      00:00  I/O WAIT TIME           0  INITIAL RUNS
         16  COMPARES                0  MERGE ORDER
          0  SCRATCH DISK
          0  SCRATCH SEEKS
Errors detected: 0
Warnings detected: 0
```

Specifying Input Records at the Input Prompt

If you do not specify input records in a command file or input files with the FROM command, FastSort prompts you to enter input records from your terminal. The input prompt is a question mark:

```
?
```

When you specify records at the input prompt, enter only one record after each input prompt. When you finish entering records, press Ctrl-Y, the logical end-of-file character:

```
16> SORT
FastSort - T9620D30 - (31OCT94)
COPYRIGHT TANDEM COMPUTERS INCORPORATED 1991 - 1994
<ASCENDING 1:10
<RUN
?weeping fig
?daffodil
?red juniper
?Ctrl-Y EOF!
daffodil
red juniper
weeping fig
          3  RECORDS               132  MAX RECORD SIZE
      00:25  ELAPSED TIME          166  BUFFER PAGES
      00:00  I/O WAIT TIME           0  INITIAL RUNS
          4  COMPARES                0  MERGE ORDER
          0  SCRATCH DISK
          0  SCRATCH SEEKS
```

Sorting on Key Fields

FastSort returns four types of output records:

Sort Operation	**Output Records**
Record Sort	The entire file of input records reordered according to the values of one or more key fields
Key Sort	The values of the concatenated key fields in sorted order
Permutation Sort	The input record sequence numbers in the order the records would be in if they were sorted according to the specified key fields
Key and Permutation Sort	A sequence number followed by the concatenated key-field values for each record

The following examples show the contents of the input files PLANTS1 and PLANTS2 with column numbers added above the records. Note where the key fields begin and end. In both files, the file contents begin with the line labeled 01.

PLANTS1 sample input file:

```
         1111111111222222222233333333334444444444555555555
1234567890123456789012345678901234567890123456789012345678
01   Aluminum       PILEA       CADIEREI      indoor    22
02   Weeping Fig    FICUS       BENJAMINA     tree      15
03   Busy Lizzy     IMPATIENS                 flower    30
04   Crocus         CROCUS                    flower    53
05   Artillery      PILEA       MICROPHYLLA   indoor    10
06   Touch-me-not   IMPATIENS                 flower    45
07   Grape Ivy      CISSUS      RHOMBIFOLIA   indoor    07
08   Rubber         FICUS       ELASTICA      tree      04
09   Fiddleleaf Fig FICUS       LYRATA        tree      01
10   Parlor Palm    CHAMAEDOREA ELEGANS       indoor    03
11   Piggy-back     TOLMIEA     MENZIESII     indoor    13
12   Daffodil       NARCISSUS                 flower    60
13   Boston Fern    NEPHROLEPIS EXALTATA      indoor    18
```

PLANTS2 sample input file:

```
         1111111111222222222233333333334444444444555555555
1234567890123456789012345678901234567890123456789012345678
01   Chinquapin Oak QUERCUS     MUEHLENBERGII tree      33
02   Aluminum       PILEA       CADIEREI      indoor    22
03   Slippery Elm   ULMUS       RUBRA         tree      10
04   Ohio Buckeye   AESCULUS    GLABRA        tree      02
05   Mesquite       PROSOPIC    JULIFLORA     shrub     48
06   Red Juniper    JUNIPERUS   VIRGINIANA    tree      14
07   American Plum  PRUNUS      AMERICANA     shrub     37
08   California Oak QUERCUS     AGRIFOLIA     tree      65
09   Red Mulberry   MORUS       RUBRA         shrub     24
10   Apple          MALUS       PUMILA        fruit     35
```

```
11   Hoptree        PTELEA      TRIFOLIATA    shrub     27
12   Catclaw Acacia ACACIA      GREGGII       shrub     12
```

Running a Record Sort

A record sort reorders input records according to the values of one or more key fields. The following commands tell FastSort to reorder the records from PLANTS1 and PLANTS2 by using the values in three key fields and to write the records to the file named SORTOUT:

```
17> SORT
FastSort - T9620D30 - (31OCT94)
COPYRIGHT TANDEM COMPUTERS INCORPORATED 1991 - 1994
<FROM PLANTS1
<FROM PLANTS2
<TO SORTOUT
<DESC 47:52
<ASC 21:31, 33:43
<RUN, REMOVEDUPS
         26  RECORDS                132  MAX RECORD SIZE
      00:04  ELAPSED TIME            63  BUFFER PAGES
      00:00  I/O WAIT TIME            0  INITIAL RUNS
        117  COMPARES                 0  MERGE ORDER
          0  SCRATCH DISK
          0  SCRATCH SEEKS
          2  DUPLICATES
```

The REMOVEDUPS parameter tells FastSort to remove any record that has the same values in all the key fields as a previous record. After the sort run, the SORTOUT file contains all the input records in sorted order, except for two records that have duplicate key-field values. FastSort preserves only the first occurrence of these records. The following example shows the contents of the SORTOUT file.

```
04   Ohio Buckeye   AESCULUS    GLABRA        tree      02
02   Weeping Fig    FICUS       BENJAMINA     tree      15
08   Rubber         FICUS       ELASTICA      tree      04
09   Fiddleleaf Fig FICUS       LYRATA        tree      01
06   Red Juniper    JUNIPERUS   VIRGINIANA    tree      14
08   California Oak QUERCUS     AGRIFOLIA     tree      65
01   Chinquapin Oak QUERCUS     MUEHLENBERGII tree      33
03   Slippery Elm   ULMUS       RUBRA         tree      10
12   Catclaw Acacia ACACIA      GREGGII       shrub     12
09   Red Mulberry   MORUS       RUBRA         shrub     24
05   Mesquite       PROSOPIC    JULIFLORA     shrub     48
07   American Plum  PRUNUS      AMERICANA     shrub     37
11   Hoptree        PTELEA      TRIFOLIATA    shrub     27
10   Parlor Palm    CHAMAEDOREA ELEGANS       indoor    03
07   Grape Ivy      CISSUS      RHOMBIFOLIA   indoor    07
13   Boston Fern    NEPHROLEPIS EXALTATA      indoor    18
01   Aluminum       PILEA       CADIEREI      indoor    22
05   Artillery      PILEA       MICROPHYLLA   indoor    10
11   Piggy-back     TOLMIEA     MENZIESII     indoor    13
10   Apple          MALUS       PUMILA        fruit     35
04   Crocus         CROCUS                    flower    53
```

```
03   Busy Lizzy     IMPATIENS                        flower    30
12   Daffodil       NARCISSUS                        flower    60
```

Running a Key Sort

The output records from a key sort consist of the values of the concatenated key fields in sorted order. The following commands tell FastSort to reorder the records in PLANTS1 and PLANTS2 using the same key fields and to write only the key values to SORTOUT:

```
18> SORT
FastSort - T9620D30 - (31OCT94)
COPYRIGHT TANDEM COMPUTERS INCORPORATED 1991 - 1994
<FROM PLANTS1
<FROM PLANTS2
<TO SORTOUT, KEYS
<DESC 47:52
<ASC 21:31, 33:43
<RUN, REMOVEDUPS
         26  RECORDS               132  MAX RECORD SIZE
      00:04  ELAPSED TIME           63  BUFFER PAGES
      00:00  I/O WAIT TIME           0  INITIAL RUNS
        131  COMPARES                0  MERGE ORDER
          0  SCRATCH DISK
          0  SCRATCH SEEKS
          2  DUPLICATES
```

The following example shows the contents of the SORTOUT file after the sort run. SORTOUT contains only the key-field values for each input record, except for the records that have duplicate values.

```
tree  AESCULUS   GLABRA
tree  FICUS      BENJAMINA
tree  FICUS      ELASTICA
tree  FICUS      LYRATA
tree  JUNIPERUS  VIRGINIANA
tree  QUERCUS    AGRIFOLIA
tree  QUERCUS    MUEHLENBERG
tree  ULMUS      RUBRA
shrub ACACIA     GREGGII
shrub MORUS      RUBRA
shrub PROSOPIC   JULIFLORA
shrub PRUNUS     AMERICANA
shrub PTELEA     TRIFOLIATA
indoorCHAMAEDOREAELEGANS
indoorCISSUS     RHOMBIFOLIA
indoorNEPHROLEPISEXALTATA
indoorPILEA      CADIEREI
indoorPILEA      MICROPHYLLA
indoorTOLMIEA    MENZIESII
fruit MALUS      PUMILA
flowerCROCUS
flowerIMPATIENS
flowerNARCISSUS
```

Running a Permutation Sort

The output records from a permutation sort consist of the input record sequence numbers, in the order the records would be in if they were sorted according to the specified key fields. The following commands direct FastSort to reorder the records in PLANTS1 and PLANTS2 using the same key fields and to write only the sequence numbers to SORTOUT:

```
19> SORT
FastSort - T9620D30 - (31OCT94)
COPYRIGHT TANDEM COMPUTERS INCORPORATED 1991 - 1994
<FROM PLANTS1
<FROM PLANTS2
<TO SORTOUT, PERM
<DESC 47:52
<ASC 21:31, 33:43
<RUN, REMOVEDUPS
         26  RECORDS               132  MAX RECORD SIZE
      00:06  ELAPSED TIME           63  BUFFER PAGES
      00:00  I/O WAIT TIME           0  INITIAL RUNS
        131  COMPARES                0  MERGE ORDER
          0  SCRATCH DISK
          0  SCRATCH SEEKS
          2  DUPLICATES
```

The following example shows the contents of the SORTOUT file after the sort run using the FUP COPY command. Note that FUP displays the record sequence numbers in octal format.

```
20> FUP COPY SORTOUT,,OCTAL

$VOL.FS.SORTOUT  RECORD 0  KEY 0 (%0)  LEN 4   2/19/92 13:47
   0: 000000 000021                                      ....
$VOL.FS.SORTOUT  RECORD 1  KEY 1 (%1)  LEN 4
   0: 000000 000002                                      ....

$VOL.FS.SORTOUT  RECORD 2  KEY 2 (%2)  LEN 4
   0: 000000 000010                                      ....

...

$VOL.FS.SORTOUT  RECORD 23  KEY 23 (%27)  LEN 4
   0: 000000 000014                                      ....
24 RECORDS TRANSFERRED
```

Running a Key and Permutation Sort

Each output record from a combined key and permutation sort consists of a sequence number followed by the key-field values. The following commands direct FastSort to reorder the records in PLANTS1 and PLANTS2 using the same key fields and to write both the sequence numbers and the key-field values to SORTOUT:

```
21> SORT
FastSort - T9620D30 - (31OCT94)
COPYRIGHT TANDEM COMPUTERS INCORPORATED 1991 - 1994
<FROM PLANTS1
<FROM PLANTS2
<TO SORTOUT, PERM, KEYS
<DESC 47:52
<ASC 21:31, 33:43
<RUN, REMOVEDUPS
         26  RECORDS                132  MAX RECORD SIZE
      00:03  ELAPSED TIME            63  BUFFER PAGES
      00:00  I/O WAIT TIME            0  INITIAL RUNS
        131  COMPARES                 0  MERGE ORDER
          0  SCRATCH DISK
          0  SCRATCH SEEKS
          2  DUPLICATES
```

The following example shows the contents of the SORTOUT file after the sort run using the FUP COPY command.

```
$VOL.FS.SORTOUT  RECORD 0  KEY 0 (%0)  LEN 32   4/07/92 14:16
   0: 000000 000021 072162 062545  020040 040505 051503 052514
052523 020040 020107 046101  ....tree  AESCULUS   GLA
 %14: 041122 040440 020040 020040                          BRA

$VOL.FS.SORTOUT  RECORD 1  KEY 1 (%1)  LEN 32
   0: 000000 000002 072162 062545  020040 043111 041525 051440
020040 020040 020102 042516  ....tree  FICUS      BEN
 %14: 045101 046511 047101 020040                       JAMINA

$VOL.FS.SORTOUT  RECORD 2  KEY 2 (%2)  LEN 32
   0: 000000 000010 072162 062545  020040 043111 041525 051440
020040 020040 020105 046101  ....tree  FICUS      ELA
 %14: 051524 044503 040440 020040                        STICA

...

$VOL.FS.SORTOUT  RECORD 22  KEY 22 (%26)  LEN 32
   0: 000000 000014 063154 067567  062562 047101 051103 044523
051525 051440 020040 020040  ....flowerNARCISSUS
 %14: 020040 020040 020040 020040
23 RECORDS TRANSFERRED
```

Controlling Extended Memory

By default, FastSort tries to use enough extended memory to make at most one merge pass, depending on the size of the output file. You can specify the maximum amount of extended memory FastSort can use with the parameters of the RUN command or with SORT DEFINEs:

Parameter	**Maximum Physical Memory FastSort Uses**
MINSPACE	256 pages
AUTOMATIC	50 percent *
MINTIME	70 percent *
SEGMENT *n*	*n* pages to a maximum of 32,767 pages If VLM is on, the maximum SEGMENT size is 62,255 pages.

* Percentage of processor memory that is not locked down when SORTPROG begins.

The maximum extended segment size for sorting depends on whether the VLM option is on or off. For more information about this option, see Using VLM on page 9-10. To use VLM for interactive sort operations, you must set up a SORT DEFINE. For more information about SORT DEFINEs, see Section 7, Using SORT and SUBSORT DEFINEs.

In addition to the parameters shown above, the amount of memory FastSort uses for sorting input records depends on:

- File size (*f*) in bytes, which is the total input record count times the maximum output record length (for a permutation sort, the record length is the key length)
- Scratch block size (*b*)
- The amount of physical memory (*m*) not locked down when SORTPROG begins

The formula for determining the approximate amount of memory FastSort needs to make no intermediate merge passes or only one intermediate merge pass is:

$$MIN\left(\sqrt{\frac{b \times f}{2}} \times 1.3, m\right)$$

Table 2-1 on page 2-12 lists the specific formulas for RUN command parameters that control extended memory. You can use the formulas to determine which parameter makes the most efficient use of your resources to sort your input records.

Table 2-1. Extended Memory Used by FastSort

Parameter	File Size, in Bytes: No Merge Passes	File Size, in Bytes: One Merge Pass	Extended Memory, in Bytes
MINSPACE	≤ 100 KB		512 KB
AUTOMATIC	≤ 100 KB		512 KB
		> 100 KB	$MIN\left(\sqrt{\frac{b \times f}{2}} \times 1.3, 0.5MB\right)$
MINTIME	≤ 200 KB		$MAX(f \times 1.3, 512KB)$
		> 200 KB	$MIN\left(\sqrt{\frac{b \times f}{2}} \times 1.3, 0.7MB\right)$
SEGMENT *n*			
n = 256	≤ 100 KB		512 KB
n > 256			$MIN(n \times 2048, 0.9MB)$

MIN	=	Minimum of two values in parentheses
MAX	=	Maximum of two values in parentheses
b	=	Scratch block size
f	=	File size in bytes (total input record count times maximum input record size)
n	=	Segment size in pages

If your input files have different maximum record lengths, you might want to specify a smaller segment size. You can use the average record length rather than the maximum to compute the file size and then specify the smaller size in the SEGMENT parameter.

In a parallel sort, FastSort distributes input records to multiple processors and scratch disks. In this case, if you specify a mode of AUTOMATIC or MINTIME, you limit the extended memory segment for the distributor-collector SORTPROG process to 90 percent of the physical memory not locked down by the operating system. Specifying MINSPACE limits the extended memory segment to 256 pages. The extended segment size limit for each subsort process is the same as for serial sorting.

You can use the VLM option to increase the amount of extended memory available for sorting. The maximum extended memory segment when VLM is on is 62,255 pages. For an interactive sort, you use a SORT DEFINE to turn on VLM. For more information, see Using VLM on page 9-10. For more information about using a SORT DEFINE, see Section 7, Using SORT and SUBSORT DEFINEs.

FastSort clears the command specifications after a run. Therefore, these specifications are in effect for only one run unless you save them. Use the SAVE and CLEAR commands to perform consecutive runs that involve similar data records and the same key fields. You can retain repeated information, and you can remove extraneous commands quickly after each run.

Understanding Statistics

After a sort or merge run, FastSort returns statistics to the list file. The list file is either the OUT file specified in the implicit TACL RUN command for the SORT process or your home terminal if you do not specify an OUT file. Following is an example of FastSort statistics:

```
         7  RECORDS                   132  MAX RECORD SIZE
     00:07  ELAPSED TIME              166  BUFFER PAGES
     00:00  I/O WAIT TIME               0  INITIAL RUNS
        16  COMPARES                    0  FIRST MERGE ORDER
         0  SCRATCH DISK                0  MERGE ORDER
         0  SCRATCH SEEKS               0  INTERMEDIATE PASSES
                                        0  NUMBER OF DUPLICATES
Errors detected: 0
Warnings detected: 0
```

The following table lists FastSort statistics:

Statistic	**Definition**
RECORDS	The number of records sorted or merged
ELAPSED TIME	The time SORTPROG took to process the sort or merge run
I/O WAIT TIME	The time SORTPROG used for calls to READ, WRITE, and AWAITIO
COMPARES	The number of times SORTPROG compared two records
SCRATCH DISK	The number of bytes in the scratch file
SCRATCH SEEKS	The number of blocked read and write operations on the scratch file
MAX RECORD SIZE	The maximum record size in bytes
BUFFER PAGES	The number of 1,024-word pages of memory SORTPROG used
INITIAL RUNS	The number of runs generated by the first pass
FIRST MERGE ORDER	The number of runs merged in the first intermediate pass
MERGE ORDER	The maximum number of runs that can be merged
INTERMEDIATE PASSES	The number of merge cycles between the initial run formation and the final merge pass
NUMBER OF DUPLICATES	The number of records with duplicate keys removed

For a parallel sort run, FastSort returns some statistics that apply only to the distributor-collector process. FastSort returns other statistics that are totals from the distributor-collector process and all subsort processes as shown below:

FastSort Process	**Statistics**
Distributor-Collector Process Only	RECORDS, BUFFER PAGES, ELAPSED TIME, INITIAL RUNS, I/O WAIT TIME, FIRST MERGE ORDER, SCRATCH DISK, MERGE ORDER, MAX RECORD SIZE, INTERMEDIATE PASSES
Distributor-Collector and Subsort Processes	COMPARES, SCRATCH SEEKS, NUMBER OF DUPLICATES

Understanding Error Messages

When an error occurs during a sort or merge run, the list file shows the FastSort error message and number of errors detected. By default, the list file is your home terminal. If an error occurs in the SORTPROG process, the error stops the process. The list file also shows the file-system error code and the name of the file that caused the error, such as:

```
 *** ERROR ***  THE FROM FILE COULD NOT BE OPENED.
OPERATING SYSTEM ERROR: 11
INPUT FILE: input-file-name
```

For a parallel sort run, FastSort displays an additional line that identifies the subsort process:

```
SORT PROCESS #2: nn,nn
```

where *nn,nn* indicates the CPU and process identification number (PIN) of the subsort process.

FastSort also displays warning messages about incorrect syntax. Warnings do not interrupt the SORTPROG process. An example of a FastSort warning is:

```
*** WARNING *** Ignoring unusable string of letters - string
```

FastSort also displays both the number of errors and warnings detected:

```
Errors detected: 2
Warnings detected: 7
```

Appendix B, FastSort Error Messages lists error codes and text and explains how to recover from them.

Understanding Completion Codes

In addition to error messages, FastSort might return a completion code after a sort or merge run. Completion codes are summarized following:

Code	Explanation
1	Syntax errors occurred but are treated as warnings only. FastSort continues to accept input and returns this message: `Syntax errors/warning detected`
2	FastSort execution errors occurred. These errors include no input file. FastSort returns the associated error code and this message: `SORT execution errors detected.`
3	FastSort could not execute the sort or merge run and returns this message: `Premature process termination with fatal errors or diagnostics.`

For completion code 3, any of the following errors can occur:

- You specified an input file with a logical name instead of the actual name. FastSort returns this message:

  ```
  Wrong name of the IN file.
  Termination info: 1
  ```

- You specified a logical file name, but you did not set DEFMODE ON. FastSort returns this message:

  ```
  DEFINE processing is not enabled.
  Termination info: 2
  ```

- You specified a list file with a DEFINE name, but you did not use a corresponding DEFINE. FastSort returns this message:

  ```
  DEFINE specification for the OUT file is missing. Termination
  info: 3
  ```

- FastSort encountered an unexpected error code while processing one or more DEFINEs. If this error occurs, report it to your service provider. FastSort returns this message:

  ```
  DEFINE error occurred when processing DEFINEs. Termination
  info: 4
  ```

- You specified a list file with a DEFINE name, but the corresponding DEFINE was not a class SPOOL DEFINE. FastSort returns this message:

  ```
  OUT file specification has illegal DEFINE class. Termination
  info: 5
  ```

For more information about using DEFINEs, see Section 7, Using SORT and SUBSORT DEFINEs.

3 Using FastSort Commands

FastSort interactive commands are summarized below. This section describes these commands in alphabetic order.

Command	Description
ASCENDING	Describes the location and attributes of one or more key fields that determine an ascending sequence for output records.
CLEAR	Deletes current command parameters for all commands or a specific command.
COLLATE	Specifies a file that contains an alternate collating sequence for comparing key fields.
COLLATEOUT	Stores an alternate collating sequence table in an unstructured file.
CPUS	Specifies processors (CPUs) in which FastSort can run subsort processes.
DESCENDING	Describes the location and attributes of one or more key fields that determine a descending sequence for output records.
EXIT	Ends an interactive FastSort session (same as Ctrl-Y).
FC	Displays the last FastSort command for editing and re-execution.
FROM	Specifies the name of an input file for a sort or merge run and the exclusion mode to use to open the file, the maximum number of records in the file, the maximum length of records in the file, and whether the records in the file are already sorted.
HELP	Displays the syntax of a specific command or a list of all FastSort commands with a description of each command.
NOTCPUS	Specifies a group of processors (CPUs) that FastSort cannot use to run subsort processes for parallel sorting.
RUN	Starts a sort or merge run and optionally specifies the SORTPROG process start parameters, the allocation of required disk space, and whether duplicate records should be removed.
SAVE	Saves FastSort command parameters from a sort or merge run to reuse in subsequent runs.
SHOW	Displays the command parameters currently in effect and whether they are entered for the next sort or merge run or saved from a previous run.
SUBSORT	Specifies the parameters for a SORTPROG subsort process for a parallel sort or merge run.
TO	Specifies an output file for a sort or merge run and parameters for the file including the percentage of data slack and index slack, whether FastSort should purge and recreate an existing output file, and the type of sort run (record, permutation, or key sort).

For more information about using these commands, see Section 2, Sorting Interactively. Appendix A, FastSort Syntax Summary contains a quick reference to the command syntax.

ASCENDING Command

Use the ASCENDING command to sort or merge records so that the values of each key field specified in the command are in smallest-to-largest order. ASCENDING and DESCENDING commands can apply to the same sort run. If you apply both commands to the same run, FastSort returns the values of the DESCENDING key fields in largest-to-smallest order.

The ASCENDING command provides this information for a sort or merge run:

- Location of one or more key fields for ordering the records
- Length and order of the key fields
- Type of data in each key field

You must specify at least one ASCENDING or DESCENDING command for each sort or merge run. The DESCENDING Command on page 3-11 has the same parameters as the ASCENDING command.

```
ASC[ENDING] field [ type ] [ , field [ type ] ]...
```

field

designates the location of a key field in the record. You can specify *field* in either of two ways:

startcol : *endcol*

startcol FOR *count*

startcol

is an integer giving the beginning column number of a key field. The numbering of record columns, or bytes, begins with 1.

endcol

is an integer giving the last column number of a key field.

count

is an integer giving the length, in bytes, of a key field.

type

describes the type of data in the key field. The default type is STRING. You can specify *type* as:

STRING
UPPER
INTEGER
REAL
UNSIGNED
SIGNED LEADING EMBEDDED or SLE
SIGNED LEADING SEPARATE or SLS
SIGNED TRAILING EMBEDDED or STE
SIGNED TRAILING SEPARATE or STS

`STRING`

specifies that the key field contains unsigned alphanumeric data. STRING is the default data type.

`UPPER`

specifies that the key field contains unsigned alphanumeric data. FastSort treats all lowercase ASCII characters as uppercase characters.

`INTEGER`

means the key field contains two's complement signed binary data.

`REAL`

specifies that the key field contains data stored in Tandem floating-point number representation. The length of the key field must be either 4 or 8 bytes, and the key field must be word aligned.

`UNSIGNED`

specifies that the key field contains unsigned binary data.

`SIGNED LEADING EMBEDDED | SLE`

specifies that the key field contains signed ASCII numeric data with the sign character (+ or –) stored in the high-order bit of the first byte in the field. The key length cannot be greater than 32 bytes.

`SIGNED LEADING SEPARATE | SLS`

specifies that the key field contains signed ASCII numeric data with the sign character (+ or –) stored in the first byte of the field. The key length cannot be greater than 32 bytes.

`SIGNED TRAILING EMBEDDED | STE`

specifies that the key field contains signed ASCII numeric data with the sign character (+ or –) stored in the high-order bit of the last byte of the field. The key length cannot be greater than 32 bytes.

`SIGNED TRAILING SEPARATE | STS`

specifies that the key field contains signed ASCII numeric data with the sign character (+ or –) stored in the last byte of the field. The key length cannot be greater than 32 bytes.

Key Fields

The order in which you enter ASCENDING and DESCENDING commands affects sort output. The first command entered has the highest priority. SORTPROG starts sorting the records according to key fields specified in the first ASCENDING or DESCENDING command, then uses key fields specified in the second ASCENDING or DESCENDING command, and so on.

If two or more records have equal values in the first key field specified in an ASCENDING or DESCENDING command, the values of the second key field, if specified, determine the sorted order of the records. If the records have equal values in the second or any successive key field, the values of the next key field, if specified, determine sorted order. If all key-field values of two or more records are equal, SORTPROG writes or returns those records in the same order it received them.

If you specify the KEYS parameter of the TO command, the output records consist of only key-field values. SORTPROG concatenates the values. SORTPROG considers all records as fixed length if you specify the KEYS parameter. If a key field extends beyond the end of a variable-length record in a structured output file, SORTPROG pads the key values with blanks.

SORTPROG can compare a key field at the end of a short record if the record contains the first byte of the key value. This is true unless the field type is REAL, SIGNED TRAILING EMBEDDED, or SIGNED TRAILING SEPARATE. For comparison of these types of data, key fields must contain complete values.

You can specify up to 63 key fields for a single sort or merge run. Fields can be contiguous, noncontiguous, and overlapping. The minimum field length is one column except for the REAL data type, which is either 4 or 8 bytes. The maximum field length is the length of the record unless you otherwise define the length.

Examples

```
ASCENDING 72:80 STRING, 1 FOR 3 INTEGER
ASC 20 FOR 8 UPPER
ASC 1:10,5:20    ! Overlapping key fields, both STRING
ASC 1:10 UNSIGNED   ! Keys of a key-sequenced file
```

CLEAR Command

Use the CLEAR command to delete command parameters entered for the current sort or merge run or saved from a previous run. You can use CLEAR to delete parameters for all commands or for individual commands.

```
CLEAR { ALL                  }
      { ASC[ENDING]          }
      { COLLATE              }
      { CPUS                 }
      { DESC[ENDING]         }
      { FROM [ filename ]    }
      { KEYS                 }
      { NOTCPUS              }
      { SUBSORT              }
      { TO                   }
```

`ALL`

deletes all current command parameters.

`ASC[ENDING]`

deletes all current key-field specifications defined by ASCENDING commands.

`COLLATE`

deletes the alternate collating sequence table.

`CPUS`

deletes the CPUS command currently in effect.

`DESC[ENDING]`

deletes all current key-field specifications defined by DESCENDING commands.

`FROM [` *`filename`* `]`

deletes current parameters for the input file *`filename`*. If you omit *`filename`*, CLEAR deletes current parameters for all files named in FROM commands.

`KEYS`

deletes all current key-field specifications for both ASCENDING and DESCENDING commands.

`NOTCPUS`

deletes the NOTCPUS command currently in effect.

`SUBSORT`

deletes current parameters of all SUBSORT commands.

`TO`

deletes all parameters for the current output file.

Examples

```
CLEAR DESC
CLEAR FROM FILETEN
CLEAR TO
```

COLLATE Command

Use the COLLATE command to specify an alternate collating sequence during a sort or merge run. This enables you to define the comparison of alphanumeric key fields or string-type data.

If you do not specify an alternate collating sequence, FastSort uses the sequence of the ASCII character set to order your results. For more information, see Appendix D, ASCII Character Set.

```
COLLATE filename
```

`filename`

is the name of an EDIT file containing a list of characters assigned to the 256 byte positions.

The sequence in which FastSort reads the file determines the character or characters assigned to each byte position. For a description of the alternate collating sequence file, see the text that follows.

The COLLATE command directs FastSort to read the EDIT file indicated by *`file`* and to translate the character list into an alternate collating sequence table. You can use the COLLATEOUT command to store this alternate collating sequence table in an unstructured file that the SORTMERGESTART procedure can use.

FastSort orders characters in the assignment list in the order it reads the characters from the alternate collating sequence file. The first character or character range in the list ranks before the second character or character range, and so on.

Specifying an Alternate Collating Sequence

You can specify an alternate collating sequence in an EDIT file by assigning one or more characters to each of the 256 byte positions. The EDIT file can contain only ranges of character assignments for the collating sequence, commas to separate ranges, and comments preceded by exclamation points. The following rules apply to specifying ranges in the file:

- All the ranges together must include character assignments for exactly 256 byte positions. You can assign more than one character to the same position. You can also assign the same character to more than one position.
- The number of lines in the file is irrelevant, but you cannot put a range of characters on more than one line.
- If the file has more than one range, commas must separate the ranges.

Assignment of Characters to Byte Positions

In a range that assigns characters to byte positions, you can use any of the following.

- Alphanumeric characters, enclosed in quotation marks
- Decimal or octal numeric literals
- The THRU keyword to abbreviate the specification of a range
- The ALSO keyword to assign more than one character to a position, which makes the characters equivalent in comparisons

A range cannot include both alphanumeric characters and numeric literals.

Alphanumeric Character Assignments

You must enclose alphanumeric character assignments in quotation marks. To include a quotation mark in a character range, use two consecutive quotation marks:

```
"ABCDEFGHIJKLMNOPQRSTUVWXYZ""+*%",
```

Numeric Literal Character Assignments

The octal numeric literals include %0 to %377, and the decimal numeric literals include 0 through 255. You can combine both in a range:

```
%10 THRU 55,
```

Abbreviated and Equivalent Character Assignments

THRU abbreviates the specification of a range of characters. You must use commas to separate a range that includes THRU from other ranges:

```
"A" THRU "Z", 65 THRU 90,
```

ALSO assigns more than one character to the same character position in the collating sequence. You must use commas to separate a range that includes ALSO from other ranges:

```
"A", ALSO "a", "B", ALSO "b",
```

To have SORTPROG treat several characters as equal in comparisons, you can assign them all to the same character position, like this:

```
";", ALSO ":", ALSO 128, ALSO 129,
```

If ALSO assigns a range of characters, the number of characters in that range must equal the number of characters in the preceding range:

```
"A" THRU "Z", ALSO "a" THRU "z",
```

Ranges beginning with ALSO do not assign characters to additional byte positions. The two ranges in the preceding example assign 52 characters to 26 byte positions. A file that contains these two ranges needs assignments for 230 additional byte positions.

Only a range that begins with ALSO can assign characters to the same byte positions as characters in another range. Unless the second range begins with ALSO, two ranges that include the same character assign the character to two different byte positions. A range that overlaps another range assigns each of the overlapping characters to a different byte position.

Invalid Alternate Collating Sequence Files

SORTPROG cannot use the alternate collating sequence from a file that has any of the following:

- An incorrect character assignment
- Not enough character assignments
- Too many character assignments

When an alternate collating sequence file is invalid for any of these reasons, FastSort returns an error message. Then, FastSort uses the ASCII sequence to collate the sort or merge run.

A mixture of string and numeric data types is an incorrect assignment:

```
"a" THRU %172     ! This is an incorrect assignment.
```

Examples of Alternate Collating Sequence Files

This subsection contains two sample EDIT files that specify the same alternate collating sequence and an example of how the alternate sequence affects sorting.

Both files deviate from the ASCII collating sequence because lowercase alphabetic characters are equivalent to uppercase characters. One file contains these lines:

```
0 THRU 32, "!""#$%&'()*+,-./",
"0" THRU "9", ":" THRU "@",
"A" THRU "Z", ALSO "a" THRU "z",
"[\]^_ {|}~", 127 THRU 255
```

The other file, named ALTSEQ, contains these lines:

```
0 THRU 64,
"A" THRU "Z", ALSO "a" THRU "z",
91 THRU 96, 123 THRU 255
```

A terminal session shows the results of sorting five records in two situations:

- Without the COLLATE command, using the ASCII collating sequence
- With the COLLATE command, using the collating sequence in ALTSEQ

```
<ASC 1:10                     ! Key
<RUN
?first record                 ! Input records
?FIRST record again
?first RECORD again
?second RECORD
?SECOND record again
?EOF!
FIRST record again            ! Output records
SECOND RECORD again
first RECORD again
first record
second RECORD
...
<ASC 1:10                     ! Key
<COLLATE ALTSEQ               ! Alternate collating sequence
<RUN
?first record                 ! Input records
?FIRST record again
?first RECORD again
?second RECORD
?SECOND record again
?EOF!
first record                  ! Output records
FIRST record again
first RECORD again
second RECORD
SECOND record again
```

COLLATEOUT Command

Use the COLLATEOUT command to store the alternate collating sequence table in a 256-byte unstructured file. The COLLATE command creates this table from an EDIT file. An application process can read the unstructured file to supply the *collate-sequence-table* parameter for the SORTMERGESTART procedure.

```
COLLATEOUT filename
```

filename

is the name of the unstructured file to which COLLATEOUT writes the 256-byte alternate collating sequence table. If *filename* already exists, FastSort purges it and creates a new file with that name.

Example

```
COLLATEOUT ALTSEQ
```

CPUS Command

Use the CPUS command to specify a group of processors (CPUs) that FastSort can use to run subsort processes for parallel sorting.

```
CPUS [ ALL      ]
     [ cpu-list ]
```

`ALL`

specifies that FastSort can use any processor. ALL is the default.

cpu-list

is a list of processor numbers, separated by commas.

If a SUBSORT statement does not specify a processor for the subsort process, FastSort follows these steps to select the processor:

1. FastSort uses the processor that runs the primary disk process for the scratch file’s volume, unless the NOTCPUS command specifies that processor.
2. Otherwise, FastSort uses any processor from the processor group, including all processors you specified in the CPUS command and did not specify in the NOTCPUS command. If you did not issue a CPUS or NOTCPUS command, the group includes all processors on your system. When FastSort selects processors for subsorts, it attempts to put each process in a different processor.
3. If FastSort cannot start the subsort process in a processor it chose, FastSort selects another processor from the group and tries to start the process in the new processor. However, FastSort does not attempt to use another processor for a subsort process if the SUBSORT statement specifies a processor that is not available.

Example

```
CPUS 1,4,5
```

DESCENDING Command

Use the DESCENDING command to sort or merge records so that the values of each key field specified in the command are in largest-to-smallest order. ASCENDING and DESCENDING commands can apply to the same sort run, which causes the values of some key fields to be in smallest-to-largest order.

The DESCENDING command provides this information before a sort or merge run:

- Location of one or more key fields for ordering records
- Length and order of key fields
- Type of data in each key field

You must specify at least one DESCENDING or ASCENDING command for each sort or merge run. The ASCENDING Command on page 3-2 has the same parameters as the DESCENDING command.

```
DESC[ENDING] field [ type ] [ , field [ type ] ]...
```

field

designates the location of a key field in the record. You can specify *field* in either of two ways:

startcol : *endcol*

startcol FOR *count*

startcol

is an integer giving the beginning column number of a key field. The numbering of record columns, or bytes, begins with 1.

endcol

is an integer giving the last column number of a key field.

count

is an integer giving the length, in bytes, of a key field.

type

describes the type of data in the key field. The default type is STRING. You can specify *type* as:

STRING
UPPER
INTEGER
REAL
UNSIGNED

SIGNED LEADING EMBEDDED or SLE
SIGNED LEADING SEPARATE or SLS
SIGNED TRAILING EMBEDDED or STE
SIGNED TRAILING SEPARATE or STS

`STRING`

specifies that the key field contains unsigned alphanumeric data. STRING is the default data type.

`UPPER`

specifies that the key field contains unsigned alphanumeric data. FastSort treats all lowercase ASCII characters as uppercase characters.

`INTEGER`

means the key field contains two's complement signed binary data.

`REAL`

specifies that the key field contains data stored in Tandem floating-point number representation. The length of the key field must be either 4 or 8 bytes, and the key field must be word aligned.

`UNSIGNED`

specifies that the key field contains unsigned binary data.

`SIGNED LEADING EMBEDDED | SLE`

specifies that the key field contains signed ASCII numeric data with the sign character (+ or –) stored in the high-order bit of the first byte in the field. The key length cannot be greater than 32 bytes.

`SIGNED LEADING SEPARATE | SLS`

specifies that the key field contains signed ASCII numeric data with the sign character (+ or –) stored in the first byte of the field. The key length cannot be greater than 32 bytes.

`SIGNED TRAILING EMBEDDED | STE`

specifies that the key field contains signed ASCII numeric data with the sign character (+ or –) stored in the high-order bit of the last byte of the field. The key length cannot be greater than 32 bytes.

`SIGNED TRAILING SEPARATE | STS`

specifies that the key field contains signed ASCII numeric data with the sign character (+ or –) stored in the last byte of the field. The key length cannot be greater than 32 bytes.

Key Fields

The order in which you enter DESCENDING and ASCENDING commands determines their relative significance. The first command has the highest priority. SORTPROG starts sorting the records according to the key fields specified in the first DESCENDING or ASCENDING command, then uses the key fields specified in the second DESCENDING or ASCENDING command, and so on.

If two or more records have equal values in the first key field specified in a DESCENDING or ASCENDING command, the values of the second key field, if specified, determine the sorted order of the records. If the records have equal values in the second or any successive key field, the values of the next key field, if specified, determine the sorted order. If all key-field values of two or more records are equal, SORTPROG writes or returns those records in the same order it received them.

When you specify the KEYS parameter of the TO command, the output records consist of only key-field values. SORTPROG concatenates the values. SORTPROG considers all records as fixed length if you specify the KEYS parameter. If a key field extends beyond the end of a variable-length record in a structured output file, SORTPROG pads the key values with blanks.

SORTPROG can compare a key field at the end of a short record if the record contains the first byte of the key value, unless the field type is REAL, SIGNED TRAILING EMBEDDED, or SIGNED TRAILING SEPARATE. For comparison of these types of data, key fields must contain complete values.

You can specify up to 63 key fields for a single sort or merge run. The fields can be contiguous, noncontiguous, and overlapping. The minimum field length is one column except for the REAL data type, which is either 4 or 8 bytes. The maximum field length is the length of the record unless you define the length otherwise.

Examples

```
DESCENDING 72:80 STRING, 1 FOR 3 INTEGER
DESC 20 FOR 8 UPPER
DESC 1:10,5:20       ! Overlapping key fields, both STRING
DESCENDING 28:34 SLS
DESC 1:10 UNSIGNED   ! Keys of a key-sequenced file
```

EXIT Command

Use the EXIT command to end an interactive FastSort session. Ctrl-Y also ends a FastSort session.

```
EXIT
```

FC Command

Use the FC (Fix) command to display the last FastSort command and then to repeat or edit the command.

```
FC
```

When you enter the FC command, FastSort displays the last command you typed followed by the FC prompt, a period (.), on the next line. At the FC prompt you can enter the following editing characters (in either uppercase or lowercase):

`R`

replaces characters in the FastSort command beginning with the character above the R with the text following the R

`I`

inserts the text following the I into the FastSort command

`D`

deletes the character in the FastSort command above the D

`//`

ends a text string and allows you to make more than one change to a command

For more information about the FC command, see the *Guardian User's Guide*.

FROM Command

Use the FROM command to specify an input file name for a sort or merge run and to provide FastSort with the following information:

- The exclusion mode that FastSort uses to open the file
- The maximum number of records in the file
- Whether the records in the file are already sorted
- The maximum length of records in the file

```
FROM [ in-file ] [ , EXCL[USION] mode ]...
                 [ , FILE count       ]
                 [ , MERGE            ]
                 [ , RECORD length    ]
```

`in-file`

names an input file containing records to be sorted or merged. You can enter multiple FROM commands, one for each input file for sorting or merging.

If you omit the *`in-file`* parameter, you can enter only one FROM command for a sort run, which means the input file is the command stream. If you use a command file (IN file) for input, put the input records after the RUN command, one record on each line. If you enter records at the terminal, type the input records after you enter the RUN command, one record at each ? prompt.

`EXCL[USION]` *`mode`*

specifies the exclusion mode with which FastSort opens a file. For *`mode`* you can specify SHARED, EXCLUSIVE, or PROTECTED.

`SHARED`

specifies that another process can write to the file while FastSort is reading it. FastSort reads the file sequentially, so that records inserted at positions already read are not included in the output file.

If another process is writing to the file while FastSort is reading it, the file system sometimes returns error 59 (FILE IS BAD). In this case, the input file is not necessarily corrupted, and you can retry the sort or merge run.

`EXCLUSIVE`

specifies that only FastSort can access the file.

`PROTECTED`

specifies that other processes can have only read access to the file. If you specify PROTECTED, the FROM `in-file` name cannot be the same as the TO *`out-file`* file name; otherwise, FastSort returns error 49 (INVALID EXCLUSION MODE SPECIFIED).

These are the default exclusion modes for different devices:

Device	**Exclusion Mode**
Permanent disk files	PROTECTED
Temporary disk files and terminals	SHARED
Other files (not disk files)	EXCLUSIVE

`FILE` *`count`*

specifies the maximum number of records in an input file. When input is from a source other than disk, FastSort uses *`count`* to estimate the space required for the initial scratch file.

If you omit the FILE parameter, SORTPROG determines the maximum number of records as follows:

- For a structured disk file, SORTPROG estimates the number of records in the file by looking at the end-of-file location and determining the structured overhead.

- For an unstructured disk file, SORTPROG calculates an approximate number of records in the file. The approximate number of records for an EDIT file is the end-of-file length multiplied by 2 and divided by the record length. The approximate number of records for other unstructured files is the end-of-file location divided by the record length. The default record length for all unstructured disk files is 132 bytes.
- For files other than disk files, the default is 50,000 records.

`MERGE`

indicates that the records in *`in-file`* do not need sorting before FastSort merges them with other input records. If you specify MERGE and the records are not sorted, FastSort returns sort error 15 (FILES TO BE MERGED MUST BE SORTED).

`RECORD` *`length`*

specifies the maximum input record length in number of bytes.

If *`in-file`* is a structured disk file, you can omit the RECORD parameter because the length is in the file label. If *`in-file`* is an odd unstructured file, you must specify the correct length for *`length`*.

Records are limited to 4080 bytes each. Data records in a command file are limited to 2000 bytes each. The default length for unstructured file records is 132 bytes.

Records belonging to key-sequenced files with increased limits are not supported using FastSort commands. Buffered interface of FastSort might be used to sort records belonging to key-sequenced files with increased limits. For more information about key-sequenced files with increased limits, see *Enscribe Programmer's Guide*.

Guidelines

Follow these guidelines when you use the FROM command.

Record Count

The value of *`count`* in the FILE parameter need not be the exact number of records in the input file. However, you should overestimate the number of records rather than underestimate the number.

If you underestimate the number of input records, FastSort might underestimate the size of an output file or the size of the extended segment, which can cause FastSort error 29. For more information about this error, see Appendix B, FastSort Error Messages.

Exclusion Mode and File Access

To enable another process to read the file at the same time as FastSort, specify PROTECTED in the FROM command and have the other process open the file in SHARED mode.

Record Entry

When you omit the FROM command or use a FROM command without an *`in-file`* parameter, you can supply the records from a terminal or from the command file (IN file) that starts the FastSort process.

△ **Caution.** If you specify the same file as both an input file and output file for a sort run, you can lose all the data from the input file if an error or processor failure ends the SORTPROG process.

Run Command

If you type the RUN command from a terminal, FastSort prompts you with a question mark (?) for each record. When you finish entering records, type the end-of-file character, Ctrl-Y.

In a command file (IN file), put the records after the RUN command, one record on each line. The actual end of the file indicates that there are no more records.

Examples

```
FROM $TAPE,RECORD 60,FILE 10000000
FROM MYFILE
FROM MYINPUT,FILE 1000,RECORD 80,EXCL PROTECTED,MERGE
FROM INMYFILE,FILE 2500,RECORD 80,MERGE
```

HELP Command

Use the HELP command to get help for FastSort commands. When you request information about a specific command, HELP displays the syntax of that command. If you do not specify a command, HELP displays a list of FastSort commands and a description of each command.

```
HELP [ ASC[ENDING]  ]
     [ CLEAR        ]
     [ COLLATE      ]
     [ COLLATEOUT   ]
     [ CPUS         ]
     [ DESC[ENDING] ]
     [ EXIT         ]
     [ FROM         ]
     [ HELP         ]
     [ NOTCPUS      ]
     [ RUN          ]
     [ SAVE         ]
     [ SHOW         ]
     [ SUBSORT      ]
     [ TO           ]
```

Examples

```
HELP
HELP FROM
HELP HELP
```

NOTCPUS Command

Use the NOTCPUS command to specify a group of processors that FastSort cannot use to run subsort processes.

```
NOTCPUS cpu-list
```

cpu-list

is a list of processor numbers, separated by commas.

Examples

You do not need to use a CPUS command before a NOTCPUS command because the default for the CPUS command is ALL. The following NOTCPUS command specifies a processor group including all processors except 2 and 3:

```
NOTCPUS 2,3
```

The NOTCPUS command is also useful to exclude processors you already specified in a CPUS command. This example excludes processors from a list specified in a previous run:

```
<CPUS 0,1,4,5,7,8,10,12  !Use any CPUs in this list.
 ...
<SAVE ALL                !Save all command parameters.
<RUN
 ...
<NOTCPUS 7,8        !Use any CPUs in the list except these.
<SAVE NOTCPUS
<RUN
 ...
```

RUN Command

Use the RUN command to start a sort or merge run. In RUN command options you can specify SORTPROG process start parameters, allocate necessary disk space, and indicate whether to remove records that have duplicate key values. RUN is the last command you can enter before a sort or merge run.

```
RUN [ scratch-file | scratch-vol]
    [ , AUTOMATIC                ]
    [ , BLOCK size               ]
    [ , CPU processor            ]
    [ , MEM memory               ]
    [ , MINSPACE                 ]
    [ , MINTIME                  ]
    [ , PRI priority             ]
    [ , { REMOVEDUPS | REMD }    ]
    [ , DEFINE define-name       ]
    [ , SEGMENT size             ]
    [ , PROGRAM file             ]
    [ , SWAP file                ]
    [ , NOSCRATCHON (scratch-vol [, scratch-vol]...)]
    [ , SCRATCHON (scratch-vol [, scratch-vol]...)]
```

scratch-file

is the name of an initial scratch file. If you omit the *scratch-file* and *scratch-vol* parameters, FastSort creates a scratch file on a suitable volume. If you specify an existing file, it must be unstructured. FastSort purges all data in an existing scratch file before using it. For more information about scratch, see Managing Sort Workspace on page 9-1.

scratch-vol

is the name of a volume for an initial scratch file. If you omit the *scratch-vol* and *scratch-file* parameters or if there is insufficient space for a scratch file on the volume you specify, FastSort tries to create a scratch file on a suitable volume. For more information about scratch files and scratch volumes, see Managing Sort Workspace on page 9-1.

`AUTOMATIC`

directs FastSort to limit elapsed time by using at most 50 percent (90 percent in parallel sorting) of the physical memory not locked down by the operating system. For files equal to or smaller than 100 KB, FastSort uses 256 pages for an extended memory segment and makes no merge pass. For larger files, FastSort attempts to use enough memory to make only one merge pass. If you do not specify the SEGMENT, MINSPACE, or MINTIME parameter, AUTOMATIC is the default.

The file size is the maximum number of records in all input files times the maximum record length for the output file. For more information, see the description of the FROM Command on page 3-14. For details about the amount of memory required to make only one merge pass for different file sizes, see Controlling Extended Memory on page 2-11.

If you specify AUTOMATIC, do not specify MINSPACE, MINTIME, or SEGMENT. If you specify one of these parameters with AUTOMATIC, FastSort ignores the parameter and returns a warning message.

If you specify AUTOMATIC for a distributor-collector process for parallel sorting, all of the subsort processes use AUTOMATIC, unless you override it with the SEGMENT parameter of the SUBSORT command or a SUBSORT DEFINE.

`BLOCK` *`size`*

specifies the size, in bytes, of input and output blocks for scratch files. The scratch file block size must be large enough to accept the largest input record, rounded up to the nearest even byte, plus 14 bytes of overhead.

The block size can be any multiple of 2048 up to 56 KB. The default is 56 KB.

`CPU` *`processor`*

specifies the processor (CPU) number in which the SORTPROG process should run. Because SORTPROG is a separate process, you can run it in a different processor from the one in which the SORT process is running. The default is the same processor.

`MEM` *`memory`*

exists only for compatibility with earlier sort-merge code. MEM specifies the number of memory pages allocated for the SORTPROG process. The size is always 64 pages. If you specify a value between 1 and 64 for MEM, FastSort ignores the value. If you specify an invalid value, FastSort returns an error message and does not start the sort or merge run.

`MINSPACE`

limits the size of the extended memory segment to 256 pages (512 KB). FastSort makes no merge pass or only one merge pass if the file size is equal to or less than 100 KB.

The file size is the maximum number of records in all input files times the maximum record length for the output file (see the FROM Command on page 3-14). For more information, see Controlling Extended Memory on page 2-11 for details about how file size affects the number of merge passes.

If you specify AUTOMATIC, do not specify MINTIME, MINSPACE, or SEGMENT in the same RUN command. If you specify one of these parameters with AUTOMATIC, FastSort ignores the parameter and returns a warning message.

If you specify MINSPACE for a distributor-collector process for parallel sorting, all of the subsort processes use MINSPACE, unless you override it with the SEGMENT parameter of the SUBSORT command or a SUBSORT DEFINE.

`MINTIME`

directs FastSort to minimize elapsed time by using at most 70 percent of the processor’s physical memory not locked down by the operating system. For files equal to or smaller than 200 KB, FastSort uses 256 pages for an extended memory segment or attempts to use enough memory to avoid an intermediate merge pass. For larger files, FastSort tries to use enough memory to make no more than one intermediate merge pass.

The file size is the maximum number of records in all input files times the maximum record length for the output file. (For more information, see FROM Command on page 3-14.) For more information for details about the amount of memory required to make only one merge pass for different file sizes, see Controlling Extended Memory on page 2-11.

If you specify AUTOMATIC, do not specify MINTIME, MINSPACE, or SEGMENT in the same RUN command. If you specify one of these parameters with AUTOMATIC, FastSort ignores the parameter and returns a warning message.

If you specify MINTIME for a distributor-collector process for parallel sorting, all of the subsort processes use MINTIME, unless you override it with the SEGMENT parameter of the SUBSORT command or a SORT DEFINE.

`PRI` *`priority`*

specifies a priority between 1 and 199 to assign to the SORTPROG process. The default is the operating system default priority for a process.

`{ REMOVEDUPS | REMD }`

Removes any records having key-field values that are duplicates of those in a previous output record. The statistics message at the end of the run reports the number of duplicates removed. If you specify an alternate collating sequence, FastSort determines which records have duplicate keys according to that collating sequence.

`DEFINE` *`define-name`*

is an optional 12-word array that specifies the name of a SORT DEFINE to use for the sort or merge run. For more information, see Section 7, Using SORT and SUBSORT DEFINEs.

`SEGMENT` *`size`*

specifies the size in pages of an extended memory segment for FastSort to use. The number of pages must be at least 256 and cannot exceed 90 percent of the processor's physical memory not locked down by the operating system. If VLM is on, 62,255 pages (127.5 MB) is the maximum segment size. If VLM is off, the maximum is 32,767 pages. For more information, see Using VLM on page 9-10 about the VLM option.

The default for segment size is AUTOMATIC, which is in effect if you do not specify the SEGMENT, MINSPACE, or MINTIME parameter.

If you specify AUTOMATIC, do not specify MINTIME, MINSPACE, or SEGMENT in the same RUN command. If you specify one of these parameters with AUTOMATIC, FastSort ignores the parameter and returns a warning message.

If you specify MINSPACE for a distributor-collector process for parallel sorting, all of the subsort processes use MINSPACE, unless you override it with the SEGMENT parameter of the SUBSORT command or a SORT DEFINE.

`PROGRAM` *`file`*

names a program file to run instead of the default. If you specify the PROGRAM parameter more than once in a RUN command, FastSort uses the last value for *`file`* that you specify.

`SWAP` *`file`*

specifies the volume, subvolume, and name of the swap file for the extended memory segment. The swap file must be on the local node.

If the file already exists, it must be unstructured. If you omit this parameter, FastSort creates a swap file on the scratch volume if the scratch file is local. For remote scratch files, the default swap file location is the volume where the program file is running.

`NOSCRATCHON (`*`scratch-vol`*` [, `*`scratch-vol`*`]...)`

specifies volumes that FastSort should not use for overflow scratch files. If the initial scratch volume becomes full, FastSort uses a volume not specified in the NOSCRATCHON attribute, protected by the Safeguard product, $SYSTEM, or a TMF audit trail disk for overflow scratch files. You can specify up to 32 NOSCRATCHON volumes. Note that this attribute requires up to 276 additional bytes of stack space. If you specify SCRATCHON, you cannot specify NOSCRATCHON.

Enclose NOSCRATCHON volume names in parentheses and separate the names with commas. FastSort recognizes the wild-card characters * and ? for NOSCRATCHON volume names. See the description of SCRATCHON under Setting DEFINE Attributes on page 7-2 for examples of how to use these characters.

`SCRATCHON (`*`scratch-vol`*` [, `*`scratch-vol`*`]...)`

specifies the volumes that FastSort should use for overflow scratch files. If the initial scratch volume becomes full, FastSort tries to create overflow scratch files on a SCRATCHON volume. You can specify up to 31 SCRATCHON volumes. Note that this attribute requires up to 276 additional bytes of stack space. If you specify NOSCRATCHON, you cannot specify SCRATCHON.

Enclose SCRATCHON volume names in parentheses and separate the names with commas. FastSort recognizes the wild-card characters * and ? for SCRATCHON volume names. See the description of SCRATCHON under Setting DEFINE Attributes on page 7-2 for examples of how to use these characters.

Examples

```
RUN TEMP,CPU 1,PRI 140
RUN ,CPU 2,BLOCK 28672,REMOVEDUPS
RUN ,PROGRAM SORTFAST,SEGMENT 256,BLOCK 28762, CPU 3
RUN $DATA.TEMP.SCRATCH,MINTIME,CPU 4
RUN NOSCRATCHON ($DATA2, $DATA3)
```

SAVE Command

Use the SAVE command to retain FastSort command parameters from a sort or merge run for future use. To retain the command parameters after a run, you must issue the SAVE command before you issue the RUN command. If you do not issue SAVE and RUN in this order, the command parameters are no longer in effect after the run finishes or if a warning occurs for a RUN command.

```
SAVE { ALL                  }
     { ASC[ENDING]          }
     { COLLATE              }
     { CPUS                 }
     { DESC[ENDING]         }
     { FROM [ filename ]    }
     { KEYS                 }
     { NOTCPUS              }
     { SUBSORT              }
     { TO                   }
```

`ALL`

saves all current command parameters.

`ASC[ENDING]`

saves all current key-field specifications defined by ASCENDING commands.

`COLLATE`

saves the alternate collating sequence table.

`CPUS`

saves the CPUS command currently in effect.

`DESC[ENDING]`

saves all current key-field specifications defined by DESCENDING commands.

`FROM [` *`filename`* `]`

saves the current parameters for the input file named *`filename`*. If you omit *`filename`*, the SAVE command saves the current parameters for all files named in FROM commands.

`KEYS`

saves all current key-field specifications for both ASCENDING and DESCENDING commands.

`NOTCPUS`

saves the NOTCPUS command currently in effect.

`SUBSORT`

saves current parameters of all SUBSORT commands.

`TO`

saves all parameters for the current output file.

To delete information retained by the SAVE command, use the CLEAR command.

Examples

```
SAVE FROM FILEIN
SAVE KEYS
```

SHOW Command

Use the SHOW command to display command parameters currently in effect, whether you entered them for the next sort or merge run or saved them from a previous run. SHOW does not display information for the COLLATE or COLLATEOUT command.

```
SHOW { ALL                  }
     { ASC[ENDING]          }
     { CPUS                 }
     { DESC[ENDING]         }
     { FROM [ filename ] }
     { KEYS                 }
     { NOTCPUS              }
     { SUBSORT              }
     { TO                   }
```

`ALL`

displays all current command parameters.

`ASC[ENDING]`

displays all current key-field specifications defined by ASCENDING commands.

`CPUS`

displays the CPUS command currently in effect.

`DESC[ENDING]`

displays all current key-field specifications defined by DESCENDING commands.

`FROM [` *`filename`* `]`

displays the current parameters for the input file named *filename*. If you omit *filename*, the SHOW command displays the current parameters for all files named in FROM commands.

`KEYS`

displays all current key-field specifications for both ASCENDING and DESCENDING commands.

`NOTCPUS`

displays the NOTCPUS command currently in effect.

`SUBSORT`

displays current parameters of all SUBSORT commands.

TO

displays all parameters for the current output file.

Examples

```
SHOW KEYS
SHOW FROM
```

SUBSORT Command

Use the SUBSORT command to set up the parameters for a SORTPROG subsort process for a parallel sort or merge run.

A subsort process runs under a SORTPROG distributor-collector process set up by a RUN command.

Note. Although you can specify up to 16 subsort processes, HP recommends that you specify no more than 8. Running more than 8 subsort processes can cause performance degradation for your system or the run to fail with FastSort error 22 (THE MEMORY SPACE FOR SORTING IS INSUFFICIENT).

The distributor-collector process reads the input file and distributes the input records to each subsort process named in the SUBSORT command. After the subsort processes finish sorting, the distributor-collector process merges the records from the subsort processes and then writes them to the output file.

```
SUBSORT scratch-file [ , BLOCK size    ]...
                     [ , CPU processor ]
                     [ , MEM memory    ]
                     [ , PRI priority  ]
                     [ , SEGMENT size  ]
                     [ , PROGRAM file  ]
                     [ , SWAP file     ]
```

scratch-file

is the name of an initial scratch file for the subsort process. If you specify an existing file, it must be unstructured. FastSort purges all data in an existing scratch file before using it.

BLOCK *size*

specifies the size in bytes of the input and output blocks for scratch files. The scratch file block size must be large enough to accept the largest input record, rounded up to the nearest even byte, plus 14 bytes of overhead.

The block size can be any multiple of 2048 bytes up to 56 KB. The default is 56 KB.

`CPU` *`processor`*

specifies the processor number for the subsort process. Because each subsort process is a separate SORTPROG process, you can run each process in a different processor. The default is the same processor in which the primary disk process for the scratch volume runs.

`MEM` *`memory`*

exists only for compatibility with earlier sort-merge code. MEM specifies the number of memory pages allocated for the SORTPROG process. The size is always 64 pages. If you specify a value between 1 and 64 for MEM, FastSort ignores the value. If you specify an invalid value, FastSort returns an error message and does not start the sort or merge run.

`PRI` *`priority`*

specifies the priority assigned to the SORTPROG process. The default is the same priority as the SORT process.

`SEGMENT` *`size`*

specifies the size in pages of an extended memory segment for the subsort process to use. The number of pages must be at least 256 but cannot exceed 90 percent of the processor's physical memory not locked down by the operating system. This value overrides the AUTOMATIC, SEGMENT, MINSPACE, and MINTIME parameters of the RUN command.

`PROGRAM` *`file`*

specifies a program file to run for the subsort process instead of $SYSTEM.SYS*nn*.SORTPROG.

`SWAP` *`file`*

specifies the name, including volume and subvolume, of the swap file for the extended memory segment. This swap file must be on the local node. If you omit the SWAP parameter, FastSort allocates a temporary swap file depending on whether the scratch file is local or remote:

Scratch File	**Location of Swap File**
Local	Same disk as initial scratch file
Remote	Disk where the SORTPROG program file is running

Examples

```
FROM INFILE
TO OUTFILE
ASC 1:10
SUBSORT $MOLD.SORT.SCRATCH, CPU 3, SEGMENT 128
SUBSORT $DP2.SORT.SCRATCH, CPU 4, SEGMENT 128
SUBSORT $RAT.SORT.SCRATCH, CPU 5, SEGMENT 128
RUN, CPU 0, AUTOMATIC
```

TO Command

Use the TO command to specify an output file for the sort or merge run and the following options for the run:

- The exclusion mode for the output file
- The type of the output file
- The percentage of data slack and index slack for the file
- Whether or not FastSort should purge and re-create an existing output file
- The type of sort or merge run: record, permutation, key, or a combination key and permutation

```
TO [ out-file ] [ , EXCL[USION] mode  ]
                [ , KEYS              ]
                [ , PERMUTATION       ]
                [ , TYPE file-type    ]
                [ , NOPURGE           ]
                [ , SLACK percentage  ]
                [ , DSLACK percentage ]
                [ , ISLACK percentage ]
```

out-file

is the name of the file to which FastSort writes the output records. If you omit the *out-file* parameter, the output goes to the file named in the *list-file* parameter of the command to start the FastSort process. If you do not specify *out-file* or *list-file*, the output goes to the home terminal for the FastSort process.

FastSort can send output to key-sequenced files, but not to an EDIT (file code 101) file. For more information about supported file types, see Appendix C, Using Supported File Types.

EXCL[USION] *mode*

specifies the exclusion mode that FastSort uses to open the output file. The exclusion mode can be SHARED, PROTECTED, or EXCLUSIVE.

SHARED

specifies that FastSort does not lock the output file. Other processes can write to the output file at the same time FastSort is writing its output. Thus, the final output file might not be in sorted order.

`PROTECTED`

specifies that only FastSort has read and write access to the output file.

`EXCLUSIVE`

specifies that only FastSort has write access to the output file. Other processes can have read access to the file.

These are the default exclusion modes for different devices:

Device	**Exclusion Mode**
Permanent disk files	PROTECTED
Temporary disk files and terminals	SHARED
Other files (not disk files)	EXCLUSIVE

`KEYS`

specifies that each output record be the value of all the key fields concatenated in the order of their significance. You determine this order by the sequence in which you enter ASCENDING and DESCENDING commands and specify the key fields in the commands.

If a key field extends beyond the end of a variable-length record in a structured output file, SORTPROG pads the key values with blanks. SORTPROG can compare a key field at the end of a short record if the record contains the first byte of the key value, unless the field type is REAL, SIGNED TRAILING EMBEDDED, or SIGNED TRAILING SEPARATE. For comparison of these types of data, key fields must contain complete values.

`PERM[UTATION]`

specifies that the output be 32-bit (4-byte) integers representing record sequence numbers. For example, if the sixty-third input record is the first record after sorting, the first number in the output is 63.

`TYPE` *`file-type`*

specifies the type of file created for the output records; *`file-type`* can be:

U	Unstructured
R	Relative
E	Entry-sequenced
K	Key-sequenced

To use an odd unstructured file for the output file, create the file using the FUP CREATE command or the CREATE system procedure before the sort or merge run and then do not set *`file-type`*.

`NOPURGE`

directs FastSort not to purge the output file if the file seems too small to contain all the output records. This parameter ensures that FastSort preserves the original partitioning and extents of the file. FastSort still purges the data from an existing output file, even though it does not purge the file.

When you specify NOPURGE, FastSort changes the record length to the default value of 132 bytes.

If an existing output file has a different file type than the TO command specifies or than SORTPROG uses by default, FastSort purges the existing file whether you specify NOPURGE or not. For more information, see Existing Output Files on page 3-31.

The SLACK, DSLACK, and ISLACK parameters apply only to key-sequenced output files. For other types of output files, FastSort ignores these parameters.

`SLACK` *`percentage`*

specifies the minimum percentage of slack space in both index and data blocks. You specify *`percentage`* as a value in the range {0:99}. The default is 0 slack.

`DSLACK` *`percentage`*

specifies the minimum percentage of slack space in data blocks. You specify *`percentage`* as a value in the range {0:99}. The default is the value of the SLACK parameter.

`ISLACK` *`percentage`*

specifies the minimum percentage of slack space in index blocks. You specify *`percentage`* as a value in the range {0:99}. The default is the value of the SLACK parameter.

Guidelines

Follow these guidelines when you use the TO command.

Output File Types

If *`out-file`* specifies a nonexistent disk file or if you do not specify NOPURGE, SORTPROG creates a new output file according to these rules in order:

1. SORTPROG uses the file type you specified in the TO command, if any.
2. SORTPROG uses the existing output file's type if it is a valid output file type and does not write output records to EDIT files.
3. SORTPROG uses the first input file's type if it is a valid disk file type for output and does not write output records to EDIT files.

4. If none of the above conditions exists, SORTPROG creates an entry-sequenced file.

You can use a process as an output file.

If `out-file` is a blocked tape file, SORTPROG writes only one record for each block. You can use the File Utility Program (FUP) to block the records and load the tape file. For information about FUP, see the *FUP Reference Manual.*

SORTPROG does not write output records to EDIT files.

Key-sequenced files with increased limits cannot be used as an output file. For more information about key-sequenced files with increased limits, see *Enscribe Programmer's Guide*.

The output file type can be key-sequenced. For key-sequenced files, the following rules apply:

- You can use only one sort key field, and the data type for the field must be UNSIGNED.
- The sort key field must be the same as the file's primary key field.
- You must specify the field in an ASCENDING command.

You can specify the data slack and index slack for a new or existing key-sequenced output file.

Existing Output Files

If `out-file` exists on a disk prior to the sort or merge run, FastSort purges all the data in the file before reusing the file. For FastSort to reuse an existing disk file as an output file, all of the following must be true:

- The existing file type must be the same as the output file type in effect for the run.
- The existing file size must be equal to or greater than the sum of all the input file sizes, except when you specify the NOPURGE parameter.
- The maximum record length for the existing file must be equal to or greater than the maximum output record length for the run.

If any of these required conditions does not exist, FastSort purges the existing output file and creates a new file. If you do not want FastSort to purge and recreate the file, specify the NOPURGE parameter in the TO command.

△ **Caution.** If you specify the same file as both an input file and output file for a sort run, you can lose all the data from the input file if an error or processor failure ends the SORTPROG process.

Output Options

If you use both PERMUTATION and KEYS, the output for each record is a 32-bit (4-byte) record number followed by the concatenated key-field values as shown in below:

```
 Byte  0 1 2 3 4 5 ...

       seq no | key 1 ...| key 2 ...
```

The first 11 characters are sequence numbers, and the remaining characters are the defined keys.

These options increase efficiency when you need only part of the data in the records. They show the permutation of the sorted records and the values of the records' key fields. If you do not specify PERMUTATION or KEYS, the output is entire records.

When printing the output to *`list-file`*, FastSort does not convert nonprintable bytes in a record. Therefore, a sorted binary integer field might not display useful information.

If you specify the PERM parameter and omit *`out-file`*, FastSort always converts the sequence numbers to the 11-digit ASCII display equivalent (10 digits and one trailing blank). The ASCII equivalent of the numeric data is packed into as few lines as possible, allowing for the line width of the output device.

Examples

```
TO SORTED,TYPE R, EXCL PROTECTED
TO ,PERMUTATION
TO PARTNOS,KEYS,PERM
TO $TAPE,EXCL EXCLUSIVE
```

4 Sorting Programmatically

The FastSort programmatic interface consists of the FastSort system procedures. You can use the FastSort system procedures to sort and merge records from an application program. You can call FastSort system procedures from an application written in any language that can call TAL procedures.

An application calls FastSort system procedures to start, control, and end a SORTPROG process. The application sets up the sort or merge run in procedure calls and supplies the input records directly or from one or more files. The SORTPROG process performs all sorting and merging operations and either writes the output records to a file or returns them to the application.

This section explains how to use a SORTPROG process from an application program. It also provides COBOL85 and TAL examples of serial sorting. For general information about sorting such as sorting on key fields, controlling extended memory size, understanding statistics, and understanding error messages, see Section 2, Sorting Interactively.

Using FastSort System Procedures

Table 4-1 lists FastSort procedures in the order in which your application might call them. For a sort or merge run, you must call SORTMERGESTART and either SORTMERGESTATISTICS or SORTMERGEFINISH. Other procedures communicate information to the SORTPROG process and return error information. For information about each procedure, including syntax, see Section 5, Using FastSort System Procedures.

Table 4-1. FastSort System Procedures (page 1 of 2)

Procedure Name	Description
SORTBUILDPARM	Specifies parameters for parallel sorting, record blocking, and overflow scratch volumes.
SORTMERGESTART	Begins the SORTPROG process and passes sort or merge parameters from the calling process to SORTPROG.
SORTMERGESEND	Sends input records from the calling process to the SORTPROG process, one for each call.
SORTMERGERECEIVE	Returns output records from the SORTPROG process to the calling process, one for each call.
SORTERROR	Provides the message text for the last FastSort error code returned by a procedure.
SORTERRORDETAIL	Provides the FastSort error code for the most recent error and if an input file caused the error, identifies the input file.

Table 4-1. FastSort System Procedures (page 2 of 2)

Procedure Name	Description
SORTERRORSUM	Provides SORTERROR and SORTERRORDETAIL information and identifies the cause of the most recent error.
SORTMERGESTATISTICS	Reports information about a sort or merge run and ends the run.
SORTMERGEFINISH	Ends the sort or merge run and stops the SORTPROG process.

Starting a Sort or Merge Run

Use the SORTMERGESTART procedure to start a SORTPROG process and specify parameters for a sort or merge run. SORTMERGESTART contains most of the necessary parameters for the sort or merge run, including:

- One or more input files
- An output file for sorting or merging, or both
- One or more key fields
- Removal of records that have duplicate key values
- The name of an initial scratch file and a block size for scratch file I/O
- Subsort processes for parallel sorting
- Parameters for running the SORTPROG process

You can use the SORTMERGESTART restart option to limit new process creation for each sort or merge run, and to reuse a scratch file in successive runs.

Ending a Sort or Merge Run

To end a sort or merge run, use either the SORTMERGESTATISTICS or SORTMERGEFINISH procedure. SORTMERGEFINISH also stops the SORTPROG process, but SORTMERGESTATISTICS does not.

Specifying Record Blocking and Parallel Sorting

Use the SORTBUILDPARM procedure to specify record blocking, parallel sorting, and overflow scratch volumes. The SORTBUILDPARM procedure specifies the following:

- A buffer for record blocking to reduce interprocess messages when you use SORTMERGESEND or SORTMERGERECEIVE (not valid for a merge run)
- Processors (CPUs) or the restart option for subsort processes in parallel sorting
- Volumes to either include or exclude from overflow scratch files

SORTBUILDPARM puts the parameters you specify in a sort control block, which is a global array used for storing the information. SORTMERGESTART uses the sort control block to pass the parameters to the SORTPROG process.

Allocating Scratch Space

You can have SORTPROG create initial and overflow scratch files for you. To do this, specify either no scratch file in SORTMERGESTART or a disk file that does not exist.

If you want SORTPROG to use only a single, permanent scratch file, use the formula described under Manually Creating a Scratch File on page 9-2 to calculate scratch file size. Use FUP to create the file and then specify the file to FastSort in SORTMERGESTART.

For example, if your input files have different maximum input record lengths, you might want to manually estimate initial scratch file size. Rather than the maximum output record length, multiply the number of input records by the average output record length. Then tell SORTPROG to use your estimate by setting SORTMERGESTART *flags*.<9> to 1.

Use the *scratchvols* structure in SORTBUILDPARM to specify volumes to include or exclude from overflow scratch files. For more information about scratch files, see Section 9, Optimizing Sort Performance. For more information about FUP, see the *File Utility Program (FUP) Reference Manual*.

Getting Information About a Sort or Merge Run

To return information about the sort or merge run to your application, use these procedures:

- SORTMERGESTATISTICS
- SORTERRORSUM

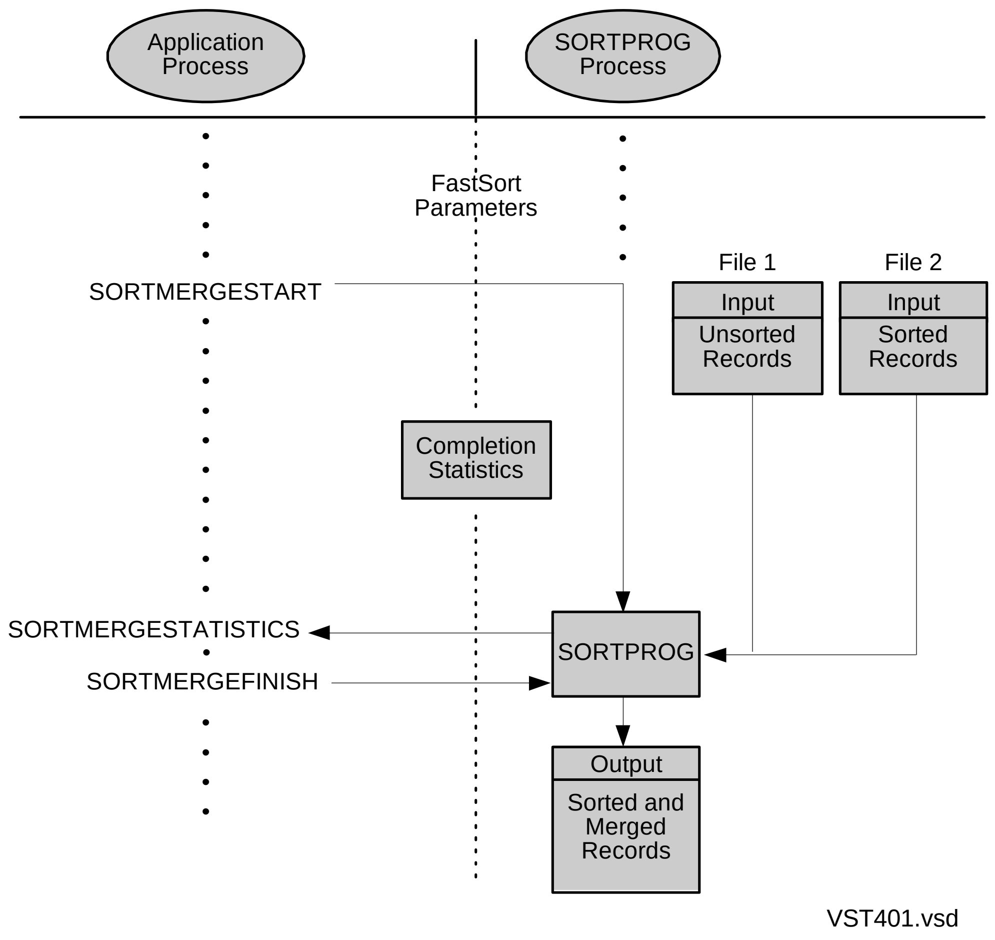
SORTMERGESTATISTICS provides details about the records and resource use after a successful run. SORTERRORSUM returns all the information provided by SORTERROR and SORTERRORDETAIL and identifies the cause of the most recent error if not an input error. For more detailed information about statistics and error messages, see Understanding Statistics on page 2-13 and Understanding Error Messages on page 2-14.

Specifying Input Records

The SORTPROG process reads records directly from one or more input files and writes the records to an output file. Each input file can contain either sorted records for merging or unsorted records for sorting and merging. When reading more than one input file, SORTPROG uses the same key-field specifications for all input records.

Figure 4-1 on page 4-4 shows sorting and merging with input and output files.

Figure 4-1. Sorting and Merging With Input and Output Files



Sending Input Records From a Process

FastSort can accept records up to 27,648 bytes in buffers of 32 KB from an application process. For input records of size greater than 4072 bytes, only buffered interface must be used.To send input records from your process to a SORTPROG process, use the SORTMERGESEND procedure as follows:

1. If you want to use record blocking, call SORTBUILDPARM.
2. Call SORTMERGESTART to start SORTPROG.
3. Call SORTMERGESEND to send each record. Specify certain parameters for a sort run or a merge run, as the following subsections explain.
4. Call SORTMERGESEND with a `length` parameter of –1 to tell SORTPROG that the last record has been sent.

Sending Records to Be Sorted

To use SORTMERGESEND for a sort run, you must specify the following values in the call to SORTMERGESTART:

- 1 for the *`num-sort-files`* parameter
- Blanks for the *`input-file-name`* parameter

Sending Records to Be Merged

To use SORTMERGESEND for a merge run, you must specify the following in the call to SORTMERGESTART:

- A number from 2 to 32 for the *`num-merge-files`* parameter
- From 2 to 32 names of all blanks for the *`input-file-name`* parameter

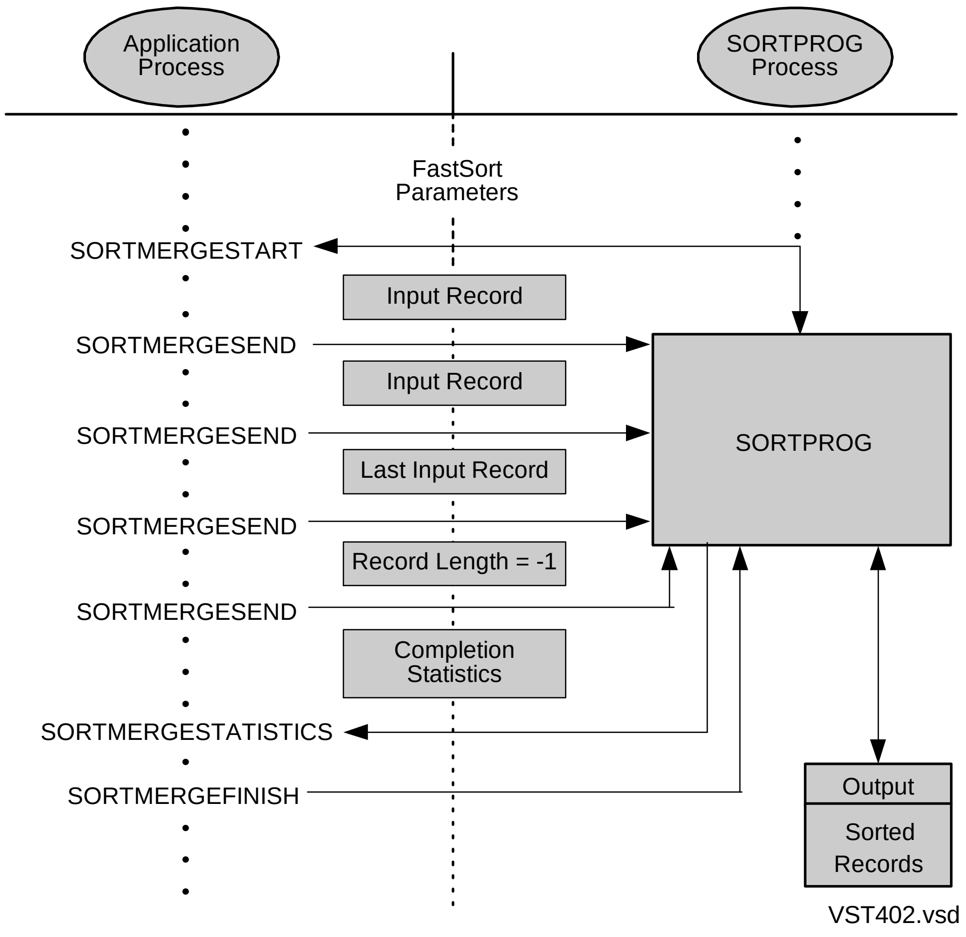
SORTPROG can merge multiple sets of records sent from a calling process. In this operation, the term input stream refers to a source of sorted records for merging. The number of input streams for merging is the number of merge files you specify in the call to SORTMERGESTART. SORTPROG merges the sorted records from all input streams into a single set of output records.

You can send records from the input streams to SORTPROG through SORTMERGESEND. The first call to SORTMERGESEND transmits a record from stream 0. Then SORTPROG returns a number in the *`stream-id`* parameter of SORTMERGESEND to indicate the input stream from which the next record should come. After SORTMERGESEND transmits all records from the input streams, SORTPROG merges the records and returns them to the calling process or produces the output file.

Record blocking is valid only for sort runs. If you try to use record blocking with merge runs, SORTPROG returns error 81 (BLOCKED INTERFACE NOT ALLOWED WITH MERGE).

Figure 4-2 on page 4-6 shows how SORTPROG accepts input records from an application process.

Figure 4-2. Sending Input Records From an Application Process



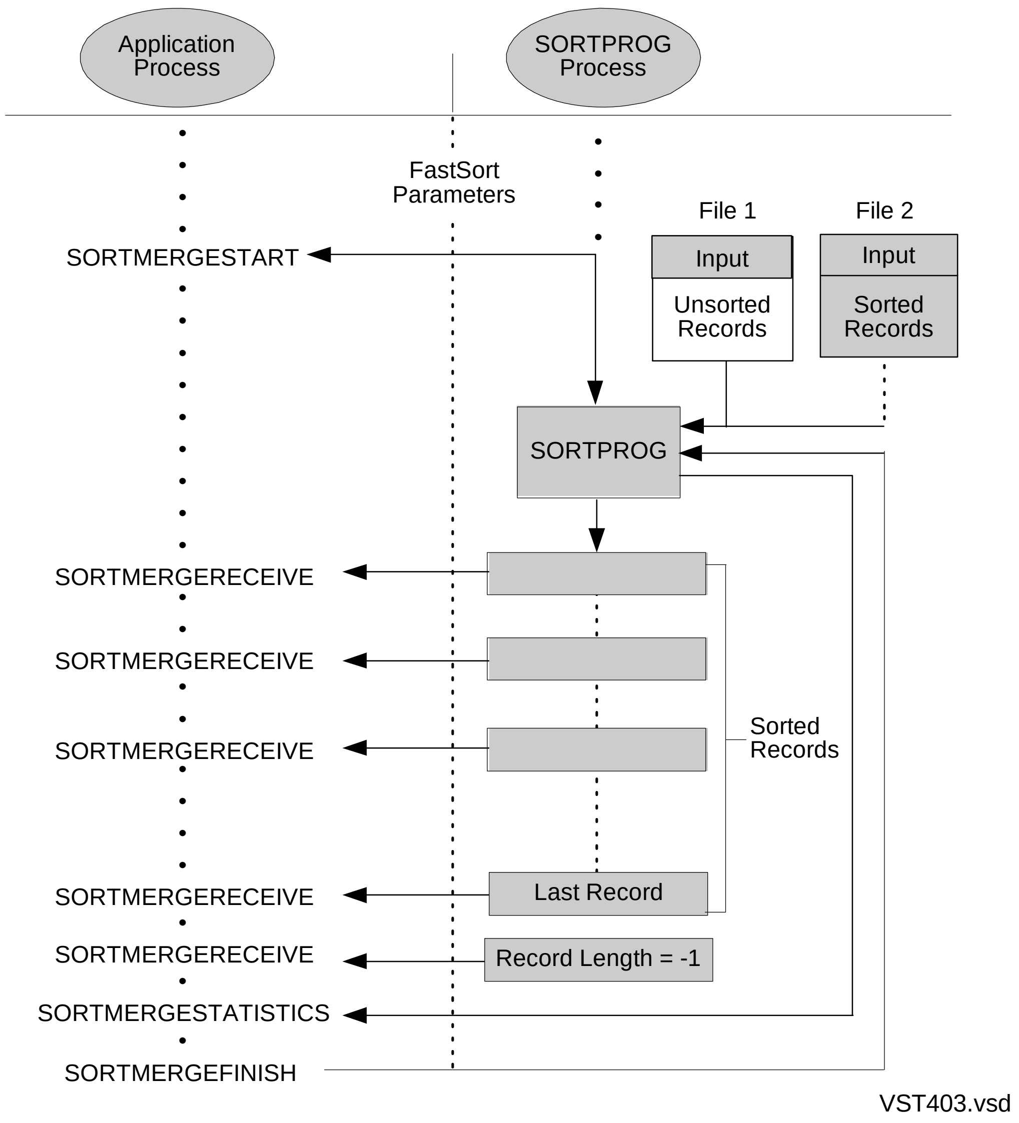
Returning Output Records to a Process

To have SORTPROG return records to your application, use the SORTMERGERECEIVE procedure as follows:

1. Call SORTMERGESTART.
2. Call SORTMERGERECEIVE to return each record until SORTPROG returns –1 in the *length* parameter of SORTMERGERECEIVE. A length of –1 means that SORTPROG has returned all the output records.

Figure 4-3 on page 4-7 shows how an application process accepts sorted records from SORTPROG.

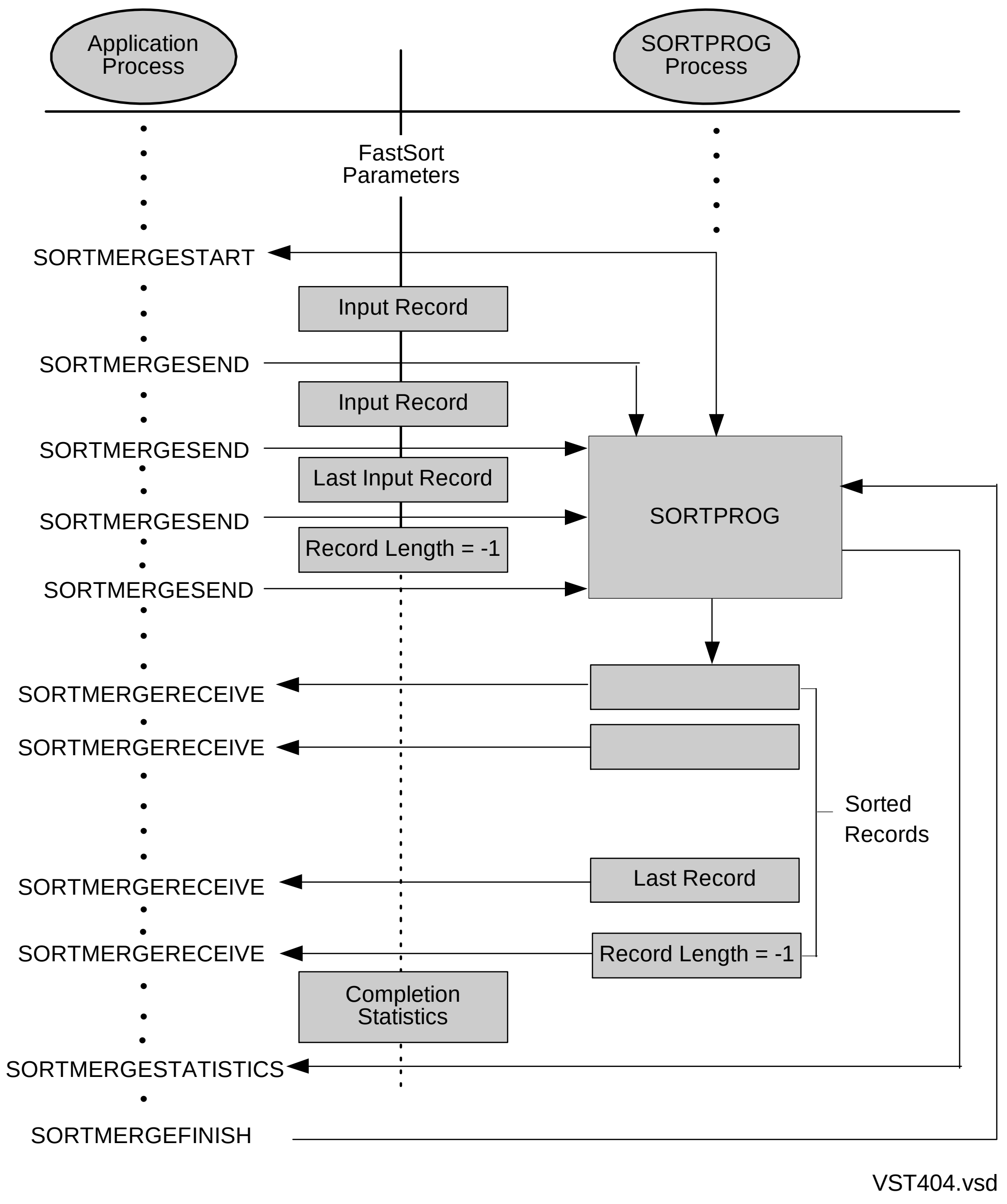
Figure 4-3. Returning Sorted Records to an Application Process



Sending and Receiving Records

Figure 4-4 on page 4-8 shows how an application process uses both SORTMERGESEND and SORTMERGERECEIVE for the same sort or merge run.

Figure 4-4. Sending and Receiving Records From an Application Process



Estimating the Size of an Output File

To estimate the size of an output file, multiply output record length by the number of input records. For structured files, allow approximately 3 percent for overhead.

If you name a new output file, FastSort estimates the size for you. If you name an existing output file, you can tell FastSort to purge the file and create a new one by specifying the *`flags`* parameter of SORTMERGESTART with *`flags`*.<14> set to 1.

You can also set *flags*.<5> to 1 to direct FastSort to not purge an existing output file that seems too small.

Sorting From C Programs

Example 4-1 shows a C program that calls FastSort procedures to perform a serial sort run.

Example 4-1. C Example of a Serial Sort Run (page 1 of 5)

```
#pragma  sql wheneverlist
#pragma  symbols
#pragma  inspect
#pragma  runnable
#pragma  nolist
/*-------------------------------------------------------------*/
/*       FastSort Serial Sort Run                              */
/*-------------------------------------------------------------*/
/* This program sends input records to a SORTPROG process      */
/* using SORTMERGESEND and then receives the sorted output     */
/* records using SORTMERGERECEIVE. To reduce interprocess      */
/* messages, this program uses a blocked interface and         */
/* declares two buffers for nowait I/O for writes to SORTPROG. */
/* Error handling and displaying of statistics are stubbed out.*/
/*-------------------------------------------------------------*/
/* External declarations                                       */
/*-------------------------------------------------------------*/
#include <stdioh>
#include <stdlibh>
#include <stringh>
#include <sqlh>
#include <talh>
#include <cextdecs>
#pragma list

#define BLOCKLEN   4096
#define MAXCOUNT   20          /* maximum record count         */
#define BUFSIZE    35          /* size of input buff array     */

char  home_term_name[48];      /* terminal name                */
short home_term_filenum;       /* file number                  */
short home_term_len;           /* actual len of hometerm name  */
short home_term_maxlen = 48;   /* max len of hometerm name     */
short error_detail;            /* output from process_getinfo_*/
/*-------------------------------------------------------------*/
/* FastSort control and flags information.                     */
/*-------------------------------------------------------------*/
_lowmem short ctlblk[200]; /* control block for sort interface */
short smflags = 0;    /* SORTMERGESTART flags                  */
short smflags2 = 1;   /* SORTMERGESTART flags2 for nowait I/0  */
short sflag1 = 1;     /* use SORTMERGESTATISTICS 22-word array */
_lowmem short key_array[4]; /* SORTMERGESTART key field defns  */
/*-------------------------------------------------------------*/
/* FastSort block buffers                                      */
/*-------------------------------------------------------------*/
long  block_buffer[BLOCKLEN - 1];
long  block_buffer2[BLOCKLEN - 1];
/*-------------------------------------------------------------*/
/* FastSort record information and buffer.                     */
/*-------------------------------------------------------------*/
long dcount = 20;             /* actual record count           */
long *pdcount = &dcount;
short inbuf[BUFSIZE];         /* record buffer                 */
_lowmem char outbuf[MAXCOUNT]; /* output buffer                */
```

Example 4-1. C Example of a Serial Sort Run (page 2 of 5)

```
/*------------------------------------------------------------*/
/* FastSort error and statistics variables.                   */
/*------------------------------------------------------------*/
short  error;                  /* error return parameter      */
_lowmem short error_buf[20],   /* error message buffer        */
              error_source[20];/* error related info          */
_lowmem long  error_code[40];  /* Fastsort & system error codes*/
struct sortstats_template {
   short maxrecordsize;
   short bufferpages;
   long  records;
   long  elapsedtime;
   long  compares;
   long  scratchseeks;
   long  iowaittime;
   long  scratchfileeof;
   long  initialruns;
   short firstmergeorder;
   short mergeorder;
   short intermediatepasses;
   long  numberofduplicates;
   } _lowmem sortstats;

void error_handler (void);
short DisplaySortStatistics (struct sortstats_template *);
*-----------------------------------------------------------*/
#pragma page  " Main logic "
/*------------------------------------------------------------*/

int main (void)
{
    short length = 2;
    short errlen = 0;
    short index;
    _lowmem short actuallen;/* for size of statistics in words */
/*------------------------------------------------------------*/
/* Perform standard initialization.                           */
/*------------------------------------------------------------*/
   error = PROCESS_GETINFO_(,,,,,,&home_term_name,
                             home_term_maxlen,
                             &home_term_len,
                             ,,,,,,,,,,,&error_detail);
   if (error)
     DEBUG;
   if (FILE_OPEN_(home_term_name,
                  home_term_len,
                 &home_term_filenum) != CCE )
     DEBUG;
   INITIALIZER;          /* read the startup message          */
/*------------------------------------------------------------*/
/* Initialize SORT key definitions array.                     */
/*------------------------------------------------------------*/
  key_array[0] = 1;   /* number of keys                       */
  key_array[1] = 2; /* definition = binary, unsigned, ascending*/
  key_array[2] = 2;   /* key length = 2 bytes                 */
  key_array[3] = 0;   /* key offset = 0 bytes                 */
```

Example 4-1. C Example of a Serial Sort Run (page 3 of 5)

```
//*----------------------------------------------------------*/
/* Call SORTBUILDPARM to initialize SORTPROG control block. */
/* Request blocked, double-buffered interface.              */
/*----------------------------------------------------------*/
  error = SORTBUILDPARM (&ctlblk[0],,,
                         &block_buffer[0], &block_buffer2[0],
                         BLOCKLEN);
  if (error)                  /* check for SORTBUILDPARM error */
     {
       errlen = SORTERRORSUM (&ctlblk[0],
                              &error_buf[0],
                              &error_code[0],
                              &error_source[0]);
     error_handler;
     return EXIT_FAILURE;
     }
/*----------------------------------------------------------*/
/* Call SORTMERGESTART to start the SORTPROG process.       */
/*----------------------------------------------------------*/
  error = SORTMERGESTART (&ctlblk[0],
                          &key_array[0],,1,,,
                          pdcount,,,,,,
                          smflags,,,,,,,,,,
                          smflags2,,);
  if (error)                 /* check for SORTMERGESTART error */
     {
       errlen = SORTERRORSUM (&ctlblk[0],
                              &error_buf[0],
                              &error_code[0],
                              &error_source[0]);
     error_handler;
     return EXIT_FAILURE;
     }
/*----------------------------------------------------------*/
/*Call SORTMERGESEND to send records to SORTPROG.Send successive*/
/* positive values to be returned in the same ascending order. */
/* Call SORTMERGERECEIVE to get sorted records from SORTPROG.  */
/*----------------------------------------------------------*/
  length = 2;        /* set length of buffer/input rec in bytes*/
                              /* size dependent on size of key */
  for (index = 1; index <= MAXCOUNT; index++)
    {
    inbuf[0] = index;      /*  set value to send to SORTPROG   */
    error = SORTMERGESEND (&ctlblk[0],,
                           length,,,,
                           (long) &inbuf[0]);
    if (error)             /* check for SORTMERGESEND error    */
       {
       errlen = SORTERRORSUM (&ctlblk[0] ,
                              &error_buf[0],
                              &error_code[0],
                              &error_source[0]);
       error_handler ;
       return EXIT_FAILURE;
       }
    }
length = -1;           /* signal end of records to be sorted  */
 error = SORTMERGESEND (&ctlblk[0],
                        ,length
                        ,
                        ,
                        ,
                        ,(long) &inbuf[0] );
```

Example 4-1. C Example of a Serial Sort Run (page 4 of 5)

```
 if (error)                  /* check  for SORTMERGESEND error */
       {
       errlen = SORTERRORSUM (&ctlblk[0],
                              &error_buf[0],
                              &error_code[0],
                              &error_source[0]);
       error_handler ;
       return EXIT_FAILURE;
       }
/*------------------------------------------------------------*/
/* Call SORTMERGERECEIVE to receive records from SORTPROG.    */
/*------------------------------------------------------------*/
   do
     {
     error = SORTMERGERECEIVE (&ctlblk[0],
                               ,
                               &length
                               ,
                               ,
                               ,(long) &inbuf[0] );

     if (error)      /* check for SORTMERGERECEIVE error       */
       {
       errlen = SORTERRORSUM (&ctlblk[0],
                              &error_buf[0],
                              &error_code[0],
                              &error_source[0]);
       error_handler;
       return EXIT_FAILURE;
       }
/*-------------------------------------------------------------*/
/* Output the values one at a time to the terminal             */
/*-------------------------------------------------------------*/
     NUMOUT (&outbuf[0],inbuf[0],10,2);
     WRITE (home_term_filenum, (short *) &outbuf[0],length);
     }
   while (length != -1);
/*-------------------------------------------------------------*/
/* Return SORTPROG completion errlen and statistics. Set       */
/* length in words, to return all statistics information.      */
/*-------------------------------------------------------------*/
  actuallen = sizeof(sortstats)/2;
  error = SORTMERGESTATISTICS (&ctlblk[0], &actuallen, &sortstats,
                               sflag1);
  if (error)          /* check for SORTMERGESTATISTICS error   */
       {
       errlen = SORTERRORSUM (&ctlblk[0],
                              &error_buf[0],
                              &error_code[0],
                              &error_source[0]);
       error_handler;
       return EXIT_FAILURE;
       }

*------------------------------------------------------------*/
/* Call function to display the statistics                     */
/*-------------------------------------------------------------*/
  error = DisplaySortStatistics (&sortstats);
  if (error)
       return EXIT_FAILURE;
```

Example 4-1. C Example of a Serial Sort Run (page 5 of 5)

```
//*--------------------------------------------------------------*/
/* Call SORTMERGEFINISH to stop SORTPROG after the process     */
/* successfully completes the current sort and merge run(s).   */
/*-------------------------------------------------------------*/
  error = SORTMERGEFINISH (&ctlblk[0]);
  if (error)                /* check for SORTMERGEFINISH error */
       {
       errlen = SORTERRORSUM (&ctlblk[0],
                             &error_buf[0],
                             &error_code[0],
                             &error_source[0]);
       error_handler;
       return EXIT_FAILURE;
       }
    FILE_CLOSE_ (home_term_filenum);
}                                      /* End of Main logic    */
void error_handler (void)
{
 /* error handling stubbed out */
  return;
}

short DisplaySortStatistics (struct sortstats_template *instats)
{
 /* Printing of statistics stubbed out */
  return EXIT_SUCCESS;
}
/*------------------------E-N-D--------------------------------*/
```

Sorting From COBOL85 Programs

When you use a SORT or MERGE statement in a COBOL85 program, COBOL85 calls FastSort procedures. COBOL85 uses SORTMERGESEND and SORTMERGERECEIVE record blocking for:

- Input procedures and output procedures specified in SORT statements
- Tape files specified in the USING phrase of SORT and MERGE statements
- Tape files or multiple output files specified in the GIVING phrase of SORT and MERGE statements

COBOL85 does not use record blocking for a program that runs as a process pair.

For tape input files, COBOL85 deblocks the records and uses the SORTMERGESEND procedure to send them to SORTPROG. For tape output files, COBOL85 blocks the records from SORTMERGERECEIVE. Instead of transferring a single record in each interprocess message between the COBOL85 program process and SORTPROG, FastSort transfers a block of input or output records in each message.

To run FastSort from a COBOL85 program, you use the COBOL85 utility library. This library resides in the $SYSTEM.SYSTEM.COBOLLIB program file. For more information about COBOLLIB, see the *COBOL85 Reference Manual*.

Example 4-2 on page 4-14 shows a COBOL85 example of a serial sort run. For a COBOL85 example of a parallel sort run, see Section 6, Sorting in Parallel.

Example 4-2. COBOL85 Example of a Serial Sort Run (page 1 of 2)

```
*---------------------------------------------------------
*            FastSort Serial Sort Run Program
*---------------------------------------------------------
* This program sorts an input file specified by the TACL
* DEFINE =INFILE and writes the sorted records to an output
* file specified by the TACL DEFINE =OUTFILE.  The program
* uses a temporary scratch file on the user's default volume.
*---------------------------------------------------------
?SYMBOLS, INSPECT
?LIBRARY $SYSTEM.SYSTEM.COBOLLIB
 IDENTIFICATION DIVISION.
   PROGRAM-ID.               FASTSORT-SERIAL-SORT.
 ENVIRONMENT DIVISION.
  CONFIGURATION SECTION.
  INPUT-OUTPUT SECTION.
    FILE-CONTROL.
    SELECT INPUT-FILE
           ASSIGN TO "=INFILE"
           ORGANIZATION IS SEQUENTIAL
           ACCESS MODE IS SEQUENTIAL.
    SELECT OUTPUT-FILE
           ASSIGN TO "=OUTFILE"
           ORGANIZATION IS SEQUENTIAL
           ACCESS MODE IS SEQUENTIAL.
    SELECT SCRATCH-FILE
           ASSIGN TO "#TEMP".
DATA DIVISION.
   FILE SECTION.
   FD INPUT-FILE
       LABEL RECORDS ARE OMITTED
       RECORD CONTAINS 25 CHARACTERS.
   01  IN-RECORD.
       05 EMPLOYEE-NAME          PIC X(20).
       05 EMPLOYEE-NUMBER        PIC 9(5).
   FD OUTPUT-FILE
       LABEL RECORDS ARE OMITTED
       RECORD CONTAINS 25 CHARACTERS.
   01  OUT-RECORD.
       05 EMPLOYEE-NAME         PIC X(20).
       05 EMPLOYEE-NUMBER       PIC 9(5).
   SD SCRATCH-FILE
       RECORD CONTAINS 25 CHARACTERS.
   01  SORT-RECORD.
       05 EMPLOYEE-NAME        PIC X(20).
       05 EMPLOYEE-NUMBER      PIC 9(5).

*---------------------------------------------------------
* Main program:  Open files and initiate SORTPROG.
*---------------------------------------------------------
 PROCEDURE DIVISION.
 OPEN-FILES.
   DISPLAY "Starting FastSort serial sort run...".
   OPEN INPUT INPUT-FILE.
   OPEN OUTPUT OUTPUT-FILE.
   SORT SCRATCH-FILE
        ON ASCENDING KEY EMPLOYEE-NAME OF SORT-RECORD,
           INPUT  PROCEDURE IS SORTIN-PROCEDURE
           OUTPUT PROCEDURE IS SORTOUT-PROCEDURE.
 DISPLAY "FastSort serial sort run completed.".
 STOP RUN.
```

Example 4-2. COBOL85 Example of a Serial Sort Run (page 2 of 2)

```
*---------------------------------------------------------
* Input:  Read input records and release to SORTPROG.
*---------------------------------------------------------
 SORTIN-PROCEDURE SECTION.
   DISPLAY "Reading input records...".
 READ-INPUT.
   READ INPUT-FILE NEXT RECORD
        AT END GO TO SORTIN-EXIT.
   RELEASE SORT-RECORD FROM IN-RECORD.
   GO TO READ-INPUT.
 SORTIN-EXIT.
   EXIT.
*---------------------------------------------------------
* Output:  Return sorted records and write to output file.
*---------------------------------------------------------
 SORTOUT-PROCEDURE SECTION.
   DISPLAY "Writing sorted records...".
RETURN-OUTPUT.
   RETURN SCRATCH-FILE
      AT END GO TO SORTOUT-EXIT.
   MOVE CORRESPONDING SORT-RECORD TO OUT-RECORD.
   WRITE OUT-RECORD.
   GO TO RETURN-OUTPUT.
 SORTOUT-EXIT.
   EXIT.
```

Sorting From TAL Programs

You can call a FastSort procedure directly from a TAL program. The program must include a declaration for the sort control block and for any variables, constants, and text identifiers you use in the procedure calls. For information about TAL declarations and the structure of TAL programs, see *TAL Reference Manual.*

Example 4-3 on page 4-16 shows a TAL program that calls FastSort procedures to perform a serial sort run.

Example 4-3. TAL Example of a Serial Sort Run (page 1 of 3)

```
?SYMBOLS, NOCODE, INSPECT, MAP, LMAP
!-----------------------------------------------------------!
!                  FastSort Serial Sort Run                 !
!-----------------------------------------------------------!
! This program sends input records to a SORTPROG process    !
! using SORTMERGESEND and then receives the sorted output   !
! records using SORTMERGERECEIVE.  To reduce interprocess   !
! messages, this program uses a blocked interface and de-   !
! clares two buffers for nowait I/O for writes to SORTPROG. !
!                                                           !
! (Note:  This program shows only FastSort procedure calls  !
! and does not contain error recovery routines or other     !
! features that might be implemented in an actual program.) !
!-----------------------------------------------------------!
! Global declarations.                                      !
!-----------------------------------------------------------!
INT .home^term^name[0:11] := 12*["  "]; ! Terminal name
INT  home^term^filenum;                 ! File number
!-----------------------------------------------------------!
! FastSort control and flags information.                   !
!-----------------------------------------------------------!
INT .ctlblk[0:199];   ! Control block for sort interface
INT  flags := 0;      ! SORTMERGESTART flags
INT  flags2 := 1;     ! SORTMERGESTART flags2 for nowait I/O
INT .key^array[0:3];  ! SORTMERGESTART key field definitions
!-----------------------------------------------------------!
! FastSort block buffers.                                   !
!-----------------------------------------------------------!
LITERAL block^length = 4096;
STRING .block^buffer [0:block^length - 1];
STRING .block^buffer2[0:block^length - 1];
!-----------------------------------------------------------!
! FastSort record information and buffer.                   !
!-----------------------------------------------------------!
LITERAL  max^count = 300;          ! Maximum count
INT(32)  dcount := 300D;           ! Record count
INT     .inbuf[0:35];              ! Record buffer
!-----------------------------------------------------------!
! FastSort error and statistics variables.                  !
!-----------------------------------------------------------!
INT     .error^buf[0:31],  ! Error message
         error^source,     ! Error related info
         error;            ! Error return parameter
INT(32)  error^code;       ! FastSort and system error codes
INT     .statistics[0:20]; ! Statistics buffer
-----------------------------------------------------------!
?PAGE "External Declarations From EXTDECS0"
!-----------------------------------------------------------!
?NOLIST
?SOURCE $SYSTEM.SYSTEM.EXTDECS0 (DEBUG,
?                                INITIALIZER,
?                                MYTERM,
?                                OPEN,
?                                SORTMERGESTART,
?                                SORTERRORSUM,
?                                SORTMERGESTATISTICS,
?                                SORTBUILDPARM,
?                                SORTMERGESEND,
?                                SORTMERGERECEIVE)
?LIST
```

Example 4-3. TAL Example of a Serial Sort Run (page 2 of 3)

```
!!------------------------------------------------------------!
?PAGE "Start of MAIN Procedure"
!------------------------------------------------------------!
PROC main^proc MAIN;
BEGIN
    INT length;
    INT index;
    INT(32) buf^addr  := $XADR(block^buffer);
    INT(32) buf^addr2 := $XADR(block^buffer2);
    INT(32) rec^addr  := $XADR(inbuf);
!------------------------------------------------------------!
! Perform standard initialization.                           !
!------------------------------------------------------------!
 CALL MYTERM (home^term^name);
 CALL OPEN (home^term^name, home^term^filenum);
   IF <> THEN CALL DEBUG;
 CALL INITIALIZER;          ! Read the startup message.
!------------------------------------------------------------!
! Initialize SORT key definitions array.                     !
!------------------------------------------------------------!
key^array[0] := 1;  ! Number of keys
key^array[1] := 9;  ! Definition = binary, signed, ascending
key^array[2] := 2;  ! Key length = 2 bytes
key^array[3] := 0;  ! Key offset = 0 bytes
!------------------------------------------------------------!
! Call SORTBUILDPARM to Initialize SORTPROG control block.  !
! Request blocked, double-buffered interface.                !
!------------------------------------------------------------!
error := SORTBUILDPARM (ctlblk,,,
                        buf^addr, buf^addr2,
                        block^length);
!------------------------------------------------------------!
! Call SORTMERGESTART to start the SORTPROG process.        !
!------------------------------------------------------------!
error := SORTMERGESTART (ctlblk,
                         key^array,,1,,,
                         dcount,,,,,,
                         flags,,,,,,,,,,
                         flags2);
IF error THEN   ! Check for SORTMERGESTART error.
   BEGIN
   length := SORTERRORSUM (ctlblk,
                           error^buf,
                           error^code,
                           error^source);
   ! Process the SORTMERGESTART error.
   END;
!------------------------------------------------------------!
! Call SORTMERGESEND to send records to SORTPROG.           !
! (Note:  This program sends successive negative values in  !
! descending order to SORTPROG.  SORTPROG then returns the  !
! values sorted in ascending order.  An actual program      !
! would get input values from another source.)             !
!------------------------------------------------------------!
```

Example 4-3. TAL Example of a Serial Sort Run (page 3 of 3)

```
length := 70;  ! Set length of buffer.
FOR index := 1 TO max^count DO
  BEGIN
  inbuf := - index;  ! Set value to send to SORTPROG.
  error := SORTMERGESEND (ctlblk,,
                          length,,,,
                          rec^addr);
  IF error THEN  ! Check for SORTMERGESEND error.
     BEGIN
     length := SORTERRORSUM (ctlblk,
                             error^buf,
                             error^code,
                             error^source);
     ! Process the SORTMERGESEND error.
     END;
  END;
 length := -1;  ! Indicate all records have been sent.
 error := SORTMERGESEND (ctlblk,,
                         length,,,,
                         rec^addr);
 IF error THEN  ! Check for SORTMERGESEND error.
    BEGIN
    length := SORTERRORSUM (ctlblk,
                            error^buf,
                            error^code,
                            error^source);
    ! Process the SORTMERGESEND error.
    END;
!------------------------------------------------------------!
! Call SORTMERGERECEIVE to receive records from SORTPROG.    !
!------------------------------------------------------------!
 DO
   BEGIN
   error := SORTMERGERECEIVE (ctlblk,inbuf,length);
   IF error THEN  ! Check for SORTMERGERECEIVE error.
      BEGIN
      length := SORTERRORSUM (ctlblk,
                              error^buf, error^code,
                              error^source);
      ! Process the SORTMERGERECEIVE error.
      END;
   END
!------------------------------------------------------------!
! Note:  At this point, an actual program would process the !
! sorted output records returned from SORTPROG.              !
!------------------------------------------------------------!
 UNTIL length = -1 ;
!------------------------------------------------------------!
! Return SORTPROG completion status and statistics. Set      !
! length to return all 21 words of statistics information.   !
!------------------------------------------------------------!
length := 21;
error := SORTMERGESTATISTICS (ctlblk, length, statistics);
  IF error THEN  ! Check for SORTMERGESTATISTICS error.
     BEGIN
     length := SORTERRORSUM (ctlblk,
                             error^buf,
                             error^code,
                             error^source);
    ! Process the SORTMERGESTATISTICS error.
    END;
END;                 ! End of MAIN Procedure !
!------------------------------------------------------------!
```

5 Using FastSort System Procedures

This section describes the FastSort system library procedures. FastSort procedures communicate between a user-written application process and a SORTPROG process. The SORTPROG process runs independently of an application process and by default resides in the $SYSTEM.SYS*nn*.SORTPROG program file.

For information about calling these procedures for serial sorting, see Section 4, Sorting Programmatically. For information about calling these procedures for parallel sorting, see Section 6, Sorting in Parallel. Both sections contain TAL and COBOL85 examples.

The table below describes the FastSort system library procedures in the order in which you call them in an application.

Procedure Name	**Description**
SORTBUILDPARM	Specifies parameters for parallel sorting, record blocking, and scratch volume structure.
SORTMERGESTART	Begins the SORTPROG process and passes parameters for a sort or merge run from the calling process to SORTPROG.
SORTMERGESEND	Sends input records from the calling process to the SORTPROG process, one for each call.
SORTMERGERECEIVE	Returns output records from the SORTPROG process to the calling process, one for each call.
SORTERROR	Provides the message text for the last FastSort error code returned by a procedure.
SORTERRORDETAIL	Provides the FastSort error code for the most recent error and, if an input file caused the error, identifies the input file.
SORTERRORSUM	Provides all information that SORTERROR and SORTERRORDETAIL provide and identifies the cause of the most recent error if not an input file.
SORTMERGESTATISTIC S	Reports information about a sort or merge run and ends the run.
SORTMERGEFINISH	Ends the sort or merge run and stops the SORTPROG process.

In addition to the 350 words required by system procedure calls, the FastSort system procedures require additional data stack space that is not automatically allocated by the BINSERV process during compilation. Use the table below to determine the amount of additional space you need to allocate for an application that calls FastSort procedures:

Operation	Description	Additional Space
Simple	Less than 5 keys, no subsorts, 1 input file	2 pages
Medium	Greater than 5 keys, either subsorts or multiple input files	3 pages
Complex	Greater than 5 keys, subsorts, multiple input files	4 pages

To allocate this additional space in an application, use one of the following methods:

- For a TAL application, use the DATAPAGES compiler directive during compilation. Specify DATAPAGES 64 to allocate the maximum amount.
- For all applications, use the Binder SET EXTENDSTACK command after compilation. Specify 64 PAGES to allocate the maximum amount.
- When you run the program, specify 64 for the MEM option of the RUN command. If you run the program from another application, specify 64 for the PROCESS_CREATE_ or NEWPROCESS[NOWAIT] *`memory-pages`* parameter.
- Move user data from the user data segment to an extended data segment to free up more data stack space for the call to SORTMERGESTART.

For information about TAL compiler directives, see the *TAL Reference Manual*. For information about the Binder SET command, see the *Binder Manual*.

In addition to the requirements listed in the table above, if you specify either the SCRATCHON or NOSCRATCHON attributes in a SORT DEFINE, FastSort requires up to 138 additional words of stack space. To learn how FastSort uses this space to build a pool of scratch volumes, see Table 5-1 on page 5-5.

If your application process starts a new process, FastSort also requires 30 to 35 additional words of stack space to support the PROCESS_CREATE_ procedure.

SORTBUILDPARM Procedure

Use SORTBUILDPARM to specify the following:

- A group of processors (CPUs) for a parallel sort run
- A buffer for record blocking
- A list of volumes to be used or not used for scratch files

The call to SORTBUILDPARM must precede the call to SORTMERGESTART. SORTBUILDPARM stores your parameters in the sort control block, and SORTMERGESTART passes the parameters to the SORTPROG process.

```
{ status := } SORTBUILDPARM ( ctlblock             ! i
{ CALL      }               ,[ cpu-mask ]          ! i
                            ,[ not-cpu-mask ]      ! i
                            ,[ buffer ]            ! i
                            ,[ buffer2 ]           ! i
                            ,[ buffer-length ]     ! i
                            ,[ build-flags ]       ! i
                            ,[ define-name ]       ! i
                            ,   reserved1          ! reserved
                            ,   reserved2          ! reserved
                            ,[ scratchvols ]      )! i
```

status returned value

INT

returns a FastSort error code if an error occurred; if not, *status* returns 0.

ctlblock input

INT:ref:200

is the same global storage array you name in the call to SORTMERGESTART. You should not rely on the information in *ctlblock*, because this information can change without warning.

cpu-mask input

INT:value

specifies processors (CPUs) in which FastSort can run subsort processes. FastSort can use a processor number (0 – 15) if the respective bit of the mask is set to on. If you omit *cpu-mask* or the call to SORTBUILDPARM, all bits of the mask are on. The *not-cpu-mask* parameter can override bit settings of *cpu-mask*.

not-cpu-mask input

INT:value

specifies the processors (CPUs) in which subsort processes cannot run. FastSort cannot use a processor number (0 – 15) if the respective bit of the mask is set to on.

buffer input

INT(32):value

is the address of a buffer that SORTPROG can use to block input records from SORTMERGESEND or deblock output records for SORTMERGERECEIVE. This buffer can be in the user data space segment (for buffer length up to 8 KB) or in an extended data segment. If the buffer is in an extended data segment, the segment must be in use at the time of the call. You should not rely on the information in *buffer*, because this information can change without warning.

For double buffering, you can also specify the *buffer2* parameter.

If you specify *buffer*, you must specify the length of *buffer* in the *buffer-length* parameter.

buffer2 input

INT(32):value

is the address of a second buffer that SORTPROG can use to block input records from SORTMERGESEND and output records for SORTMERGERECEIVE. Like *buffer*, *buffer2* can be in the user data space segment (for buffer length up to 8 KB) or in an extended data segment. If the buffer is in an extended data segment, the segment must be in use at the time of the call. Also, if you specify both buffers in an extended data segment, they must be in the same segment. You should not rely on the information in *buffer2*, because this information can change without warning.

If you specify *buffer2*, you must specify the *buffer* and *buffer-length* parameters. Both buffers have the length *buffer-length*.

Record blocking is valid only for sort runs. If you try to use record blocking with merge runs, SORTPROG returns error 81 (BLOCKED INTERFACE NOT ALLOWED WITH MERGE).

buffer-length input

INT:value

is the length, in bytes, of *buffer* and of *buffer2* (if specified). The length can range from 4 KB to 32 KB.

build-flags input

INT:value

is limited to the *build-flags*.<15> bit, which specifies the same restart option as the SORTMERGESTART restart flag (*flags*.<15>). For more information on description of *flags*.<15> bit, see Table 5-4 on page 5-32.

Other *build-flags* bits are not used and should be set to 0.

To preserve SORTBUILDPARM parameters in the sort control block when you use the restart option, call SORTBUILDPARM with *build-flags*.<15> set to 1 before you call SORTMERGESTART with *flags*.<15> set to 1. Before your process can use the restart flags, it must call SORTMERGESTATISTICS or an error must end the SORTPROG process.

`define-name` input

INT:ref:12

is an optional 12-word array that specifies the SORT DEFINE name to be used. For more information, see Section 7, Using SORT and SUBSORT DEFINEs.

`reserved1` `and` *`reserved2`*

are reserved for future parameters. If you specify a value for *`reserved1`* or *`reserved2`*, FastSort returns an error.

`scratchvols` `input`

INT:ref:*

is a pointer to an array of the form shown in Table 5-1.

Table 5-1. SORTBUILDPARM *`scratchvols`* Structure

Word	Description
0:7	Eight words reserved for use by FastSort library procedures.
8	Set to zero if SORTPROG should use specified volumes for scratch files. Set to one if SORTPROG should not use specified volumes for scratch files.
9	The number of volumes specified in the following list. The range is 1 to 32.
10:13*	The first entry on the list of volume names. The volume name must be of the form $data, $data*, $sp?o*, and so on. The volume name must be eight bytes long, blank padded on the right. There are no list separators.
:	:
:	:
:	:
137	If there are 32 volumes on the list, the final word of the array is word 137.

***** Words 14 through 137 are optional.

Using 32 KB Buffers

FastSort supports buffers up to 32 KB.

Example

```
build^status := SORTBUILDPARM (sortblock,,,
                               blockbuf,
                               dblbuff,
                               32768);
```

FastSort supports only buffered interface for records greater than 4072 bytes.

Guidelines

Follow these guidelines when you call the SORTBUILDPARM procedure.

Specifying a Group of Processors for Subsort Processes

When you configure a parallel sort run, you can have the distributor-collector SORTPROG process select processors for subsort processes. SORTPROG considers processors you specify in the *`cpu-mask`* parameter. You can use the *`not-cpu-mask`* parameter, which overrides *`cpu-mask`*, to exclude one or more processors.

FastSort selects a processor from the group if you do not specify a processor for a subsort process in the *`process-start`* parameter of SORTMERGESTART. If you do not specify any processors, FastSort puts each subsort process in the processor that runs the disk process for the subsort initial scratch file.

Improving Performance With Record Blocking and Nowait I/O

If your program calls SORTMERGESEND or SORTMERGERECEIVE, you can reduce the number of interprocess messages by using a single or double buffer for record blocking. SORTMERGESTART provides the buffer to transfer a block of records to or from SORTPROG, instead of a single record, in each interprocess message.

Record blocking is valid only for sort runs. If you try to use record blocking with merge runs, SORTPROG returns error 81 (BLOCKED INTERFACE NOT ALLOWED WITH MERGE).

Each call to SORTMERGESEND puts a record into the buffer, and each call to SORTMERGERECEIVE returns a record from the buffer. FastSort transfers blocks of unsorted records out of the buffer and blocks of sorted records into the buffer. You specify block size, from 4 KB to 32 KB, in the *`buffer-length`* parameter.

Your process can use the second buffer to send or receive a single record while FastSort transfers a block of records. This feature reduces the time your process waits for SORTMERGESEND or SORTMERGERECEIVE to complete its operation.

Use *`buffer2`* only if you want nowait I/O. You also need to use the *`flags2`* parameter in the call to SORTMERGESTART, with the *`flags2`*.<15> bit set to 1. Then the FastSort routines call AWAITIO and switch the buffers when necessary.

△ **Caution.** YIf you use nowait I/O, your process should not call AWAITIO to wait on any file (filenum=-1). If your application program calls AWAITIOX - 1 while a sort in invoked, the sort will fail with error 5 (COMMUNICATIONS WITH THE SORT PROCESS HAVE FAILED).

For more information about AWAITIO, see the *Guardian Procedure Calls Reference Manual*. For more information about nowait I/O, see the *Guardian Programmer's Guide*.

Figure 5-1 on page 5-7 shows how FastSort transfers unblocked records between your process and SORTPROG.

Figure 5-1. Sending and Receiving Unblocked Records

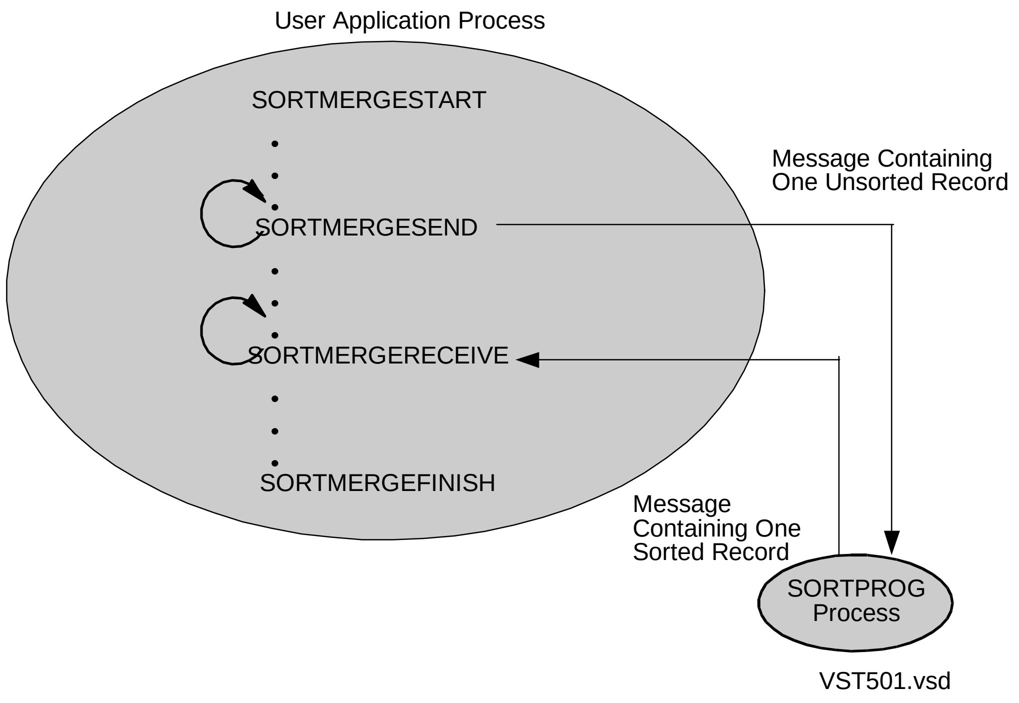
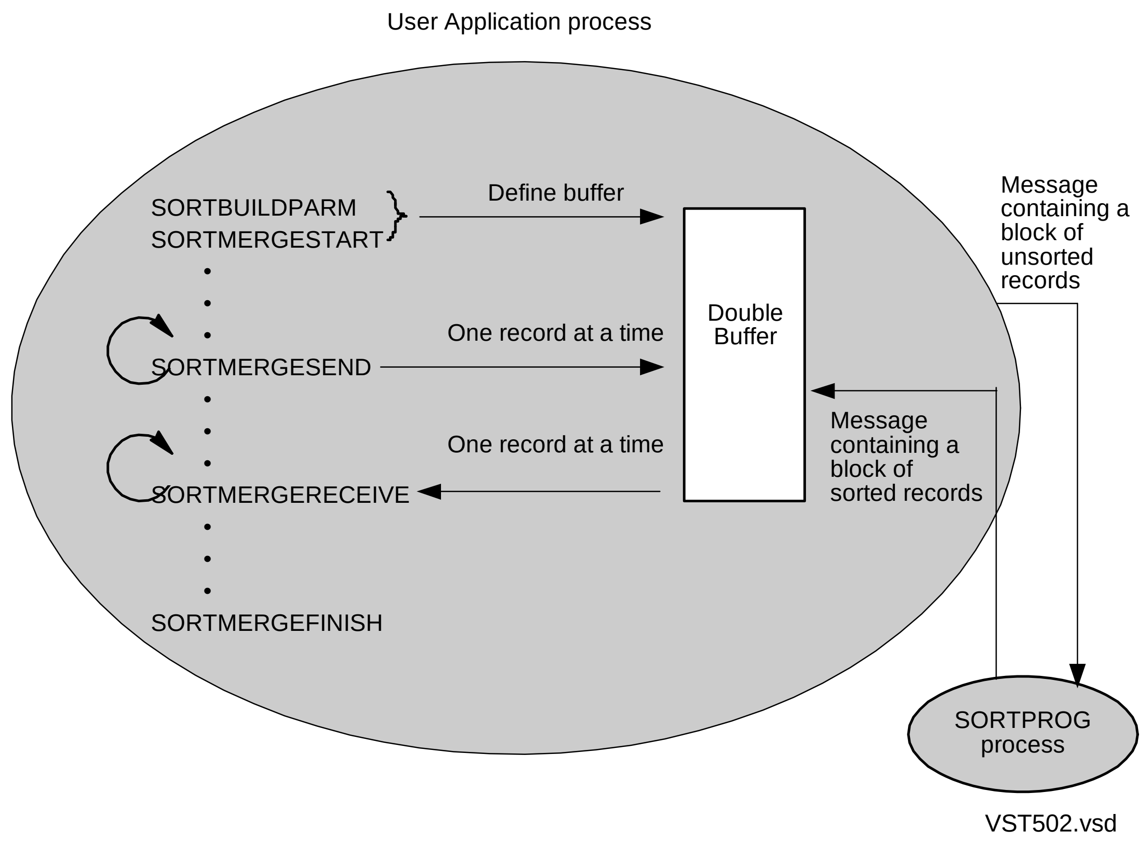


Figure 5-2 on page 5-8 shows how FastSort transfers blocked records between your process and SORTPROG if you use nowait I/O.

Figure 5-2. Sending and Receiving Blocked Records



Using Buffers in Extended Addresses

If `buffer` or `buffer2` is an extended address, the address must be relative. It cannot be an absolute extended address. The extended segment must be allocated and in use when the FastSort library procedures are called. Do not deallocate or decrease the size of the extended data segment after calling SORTBUILDPARM. An invalid extended address causes an illegal address trap.

Example

```
build^status := SORTBUILDPARM (sortblock,,,
                               blockbuf,
                               dblbuff,
                               8192);
```

SORTERROR Procedure

Use SORTERROR to provide the message text for the last FastSort error code returned by a FastSort procedure.

```
{ length := } SORTERROR ( ctlblock            ! i
{ CALL      }            , buffer  )          ! o
```

length returned value

INT

returns the number of characters in the error message.

ctlblock input

INT:ref:200

is the same global storage array you name in the call to SORTMERGESTART. You should not rely on the information in *ctlblock*, because this information can change without warning.

buffer output

INT:ref:32

is a 32-word integer array that receives the FastSort error code message text. SORTPROG does not pad the text with blanks if the buffer is shorter than 32 words. Any bytes to the right of the text remain unchanged.

Example

```
textlen := SORTERROR (sortblock,
                      outbuf);
```

SORTERRORDETAIL Procedure

Use SORTERRORDETAIL to obtain the file-system or NEWPROCESS error code and the FastSort error code for the most recent error. If an input file caused the error, SORTERRORDETAIL also uses an index to identify the file in the array of file names created by the *in-file-name* parameter of the SORTMERGESTART procedure.

```
{ status := } SORTERRORDETAIL ( ctlblock )      ! i
{ CALL      }
```

status returned value

INT(32)

returns error codes and the index of an input file in a double-word integer. The high-order word contains the file-system or NEWPROCESS error code. The low-order word contains the FastSort error code in the low-order byte and an index identifying the input file that caused the error in the high-order byte. The index is one of those in the array of file names created by the *`in-file-name`* parameter of SORTMERGESTART.

If no input file caused the error or if no error is outstanding, the low-order bits 0 through 7 are 0.

This is the format for the double-word integer:

Parameter Word	Bits 0 1 2 3 4 5 6 7	8 9 10 11 12 13 14 15
High-order	File system or NEWPROCESS(NOWAIT) error code	
Low-order	FastSort input file index	FastSort error code

`ctlblock` `input`

INT:ref:200

is the same global storage array you name in the call to SORTMERGESTART. You should not rely on the information in *`ctlblock`*, because this information can change without warning.

Example

```
detail^status := SORTERRORDETAIL (sortblock);
```

SORTERRORSUM Procedure

Use SORTERRORSUM to obtain the information that SORTERROR and SORTERRORDETAIL provide and to identify the cause of the last error. In parallel sorting, SORTERRORSUM specifies the process and processor (CPU) in which the last error occurred.

```
{ length := } SORTERRORSUM ( ctlblock           ! i
{ CALL     }                ,[ buffer ]         ! o
                            ,[ error-code ]     ! o
                            ,[ error-source ]   ! o
                            ,[ subsort-index ]  ! o
                            ,[ subsort-id ] )   ! o
```

`length` `returned value`

INT

returns the number of characters in the error message.

`ctlblock` `input`

INT:ref:200

is the same global storage array you name in the call to SORTMERGESTART. You should not rely on the information in *`ctlblock`*, because this information can change without warning.

`buffer` `output`

INT:ref:*

is a 16-word integer array that receives the error message text. SORTPROG does not pad the text with blanks if the buffer is shorter than 16 words. Any bytes to the right of the text remain unchanged.

`error-code` `output`

INT(32)

receives error codes and the index of an input file in a double-word integer. The high-order word contains the file-system or NEWPROCESS error code. The low-order word contains the FastSort error code in the low-order byte and an index identifying the input file that caused the error in the high-order byte. The index is one of those in the array of file names created by the *`in-file-name`* parameter of the SORTMERGESTART procedure.

If no input file caused the error or if no error is outstanding, the low-order bits 0 through 7 are 0.

This is the format for the double-word integer:

Parameter Word	**Bits**															
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
High-Order	**File-System or NEWPROCESS[NOWAIT] Error Code**															
Low-Order	**FastSort Input File Index**							**FastSort Error Code**								

VST503.vsd

`error-source` `output`

INT:ref:1

is a buffer that specifies the cause of the last FastSort error; *`error-source`* can be one of the following values:

Value	**Cause of Error**
–1	The information is not available.
1	Input file
2	Output file

Value	**Cause of Error**
3	Scratch file
4	The free-list file (an additional scratch file that SORTPROG allocates for internal memory management when sorting large amounts of data)
5	Process communication

`subsort-index`   output

INT:ref:1

receives the relative number of a subsort process that caused the last error. If the distributor-collector caused the last error or if you did not specify any subsort processes, SORTPROG sets *subsort-index* to –1.

`subsort-id`   output

INT:ref:1

receives the CPU and process identification numbers (PINs) of the subsort process that caused the last error. If you did not specify a subsort processes or if the distributor-collector process caused the error, SORTPROG sets *subsort-id* to –1.

Example

```
error^length := SORTERRORSUM (sortblock,
                              buffer,
                              error^code,
                              source,
                              subsort^index,
                              subsort^id);
```

SORTMERGEFINISH Procedure

Use SORTMERGEFINISH to end the SORTPROG process after the process completes the sort or merge run. A sorting or merging error stops the SORTPROG process when the error occurs. If SORTPROG stops due to an error, the next call to a FastSort procedure returns an error.

```
{ status := } SORTMERGEFINISH ( ctlblock       ! i
{ CALL      }                  ,[ abort ]      ! i
                               ,[ spare1 ]     ! reserved
                               ,[ spare2 ] )   ! reserved
```

`status`   returned value

INT

returns a FastSort error code if an error occurred; if not, returns 0. For more information about error messages, see Appendix B, FastSort Error Messages.

`ctlblock` input

INT:ref:200

is the same global storage array you name in the call to SORTMERGESTART. You should not rely on the information in *ctlblock*, because this information can change without warning.

`abort` input

INT:value

specifies when the SORTPROG process should stop:

0 Specifies that the SORTPROG process stop after completion of the current sort or merge run. This is the default value.

1 Specifies that the SORTPROG process stop immediately. The calling process receives system message -5 in its $RECEIVE file:

```
-5 PROCESS NORMAL DELETION (STOP)
```

spare1 and *spare2* reserved

are reserved for future parameters. If you specify a value for *spare1* or *spare2*, FastSort returns an error.

Example

```
error := SORTMERGEFINISH (sortblock);
```

SORTMERGERECEIVE Procedure

Use SORTMERGERECEIVE to return the output records from the SORTPROG process directly to the calling process. Use SORTMERGERECEIVE if you omit the *out-file-name* parameter from the call to SORTMERGESTART or if *out-file-name* equals all blanks.

```
{ status := } SORTMERGERECEIVE ( ctlblock          ! i
{ CALL      }                   ,[ record-loc ]     ! o
                                ,length             ! o
                                ,[ spare1 ]         !
reserved
                                ,[ spare2 ]         !
reserved
                                ,[ record-loc-ext ] ! o
```

status returned value

INT

returns a FastSort error code if an error occurred; if not, returns 0.

`ctlblock` `input`

INT:ref:200

is the same global storage array you name in the call to SORTMERGESTART. You should not rely on the information in *`ctlblock`*, because this information can change without warning.

`record-loc` `output`

INT:ref:*

is a memory location for receiving a record. The maximum record size from SORTMERGESTART determines the maximum length of this buffer. You must specify *`record-loc`* or *`record-loc-ext`*, but you cannot specify both. For buffer size of 32 KB, the *`record-loc`* cannot be used, instead *`record-loc-ext`* must be used.

`length` `output`

INT:ref:1

receives the length, in bytes, of the record retrieved. A value of –1 indicates there are no more records to return.

`spare1` `and` *`spare2`* `reserved`

are reserved for future parameters. Specifying a value for *`spare1`* or *`spare2`* causes an error. However, if you specify *`record-loc-ext`*, you must put the commas in the call to reserve places for these parameters.

`record-loc-ext` `output`

INT(32):ref:*

is an extended memory location for receiving a record. You must specify *`record-loc-ext`* or *`record-loc`*, but you cannot specify both parameters. For buffer size of 32 KB, only the *`record-loc-ext`* must be used.

Guidelines

Follow these guidelines when you call the SORTMERGERECEIVE procedure.

Omitting the Name of the Output File

If you omit the *`out-file-name`* parameter from the call to SORTMERGESTART or if *`out-file-name`* equals all blanks, you must call SORTMERGERECEIVE to return the output records, one for each call, to the calling process. You specify the format for the output records in the *`format`* parameter of SORTMERGESTART. SORTPROG produces output records in any of these formats:

- The entire record

- The sequence number as a 32-bit (4-byte) integer (permutation sort)
- The key-field values strung together (key sort)
- The sequence number followed by the key-field values strung together (permutation and key sort)

Receiving Output Records in Extended Memory

You can receive output records from SORTPROG in an extended data segment (which must be in use at the time of the call). If you want SORTMERGERECEIVE to return a record to a location in extended memory (which must be mapped), use the *record-loc-ext* parameter instead of *record-loc* to specify the address.

Deblocking Records to Reduce Interprocess Messages

You can specify a single or double buffer for record blocking and deblocking in a call to SORTBUILDPARM. SORTMERGESTART provides the buffer for FastSort to transfer a block of records (instead of a single record) in each interprocess message to or from SORTPROG. Each call to SORTMERGERECEIVE returns a record from this buffer.

Record blocking is valid only for sort runs. If you try to use record blocking with merge runs, SORTPROG returns error 81 (BLOCKED INTERFACE NOT ALLOWED WITH MERGE).

Example

This example specifies RECLOC to receive the output record and LENGTH to receive the number of bytes in the record returned:

```
receive^status := SORTMERGERECEIVE (sortblock,
                                    recloc,
                                    length);
```

SORTMERGESEND Procedure

Use SORTMERGESEND to provide input records from the calling process directly to the SORTPROG process. Use SORTMERGESEND if you omit the *in-file-name* parameter from the call to SORTMERGESTART or if *in-file-name* equals all blanks.

```
{ status := } SORTMERGESEND ( ctlblock           ! i
{ CALL      }                ,[ record-loc ]     ! i
                             ,length             ! i
                             ,[ stream-id ]      ! o
                             ,[ spare1 ]         ! reserved
                             ,[ spare2 ] )       ! reserved
                             ,[ record-loc-ext ] ! i
```

`status` returned value

INT

returns a FastSort error code if an error occurred; if not, returns 0.

`ctlblock` input

INT:ref:200

is the same global storage array you name in the call to SORTMERGESTART. You should not rely on the information in *`ctlblock`*, because this information can change without warning.

`record-loc` input

INT:ref:*

is the memory location of an input record. You must specify *`record-loc`* or *`record-loc-ext`*, but you cannot specify both. *`record-loc-ext`* must be used instead of *`record-loc`* for records of size greater than 4072 bytes.

`length` input

INT:value

is the length, in bytes, of the input record. The length can vary for input records for a sort or merge run. The length can be no smaller than the offset from the start of a record to the first character of the rightmost key and no larger than the longest input record length you specify in the *`in-file-record-length`* parameter of the SORTMERGESTART procedure.

After sending the last record for a sort run, call SORTMERGESEND with *`length`* set to –1, which indicates to SORTPROG that your process has sent all the records for the run.

After sending the last record from an input stream for a merge run, call SORTMERGESEND with *`length`* set to –1 for the stream, which indicates to SORTPROG that your process has sent all the records from the input stream.

`stream-id` output

INT:ref:1

receives the number of the input stream from which SORTMERGESEND should get the next record for merging. When all input streams have no more records, SORTMERGESEND sets *`stream-id`* to –1 to indicate that all input was sent.

You must specify this parameter if you specify more than one merge file in the call to SORTMERGESTART and if *`in-file-name`* equals all blanks.

`spare-1` `and` *`spare-2`* `reserved`

are reserved for future parameters. If you specify a value for *`spare1`* or *`spare2,`* FastSort returns an error. However, if you specify *`record-loc-ext`*, you must put the commas in the call to reserve places for these parameters.

record-loc-ext `input`

INT(32)

is the extended memory location of an input record. You must specify *record-loc-ext* or *record-loc*, but you cannot specify both parameters. *record-loc-ext* must be used instead of *record-loc*, for records of size greater than 4072 bytes.

Guidelines

Follow these guidelines when you call the SORTMERGESEND procedure.

Omitting the Input File Name or Names

If you omit the *in-file-name* parameter from the call to SORTMERGESTART or if *in-file-name* equals all blanks, you must call SORTMERGESEND to provide records for sorting or merging. Each call to SORTMERGESEND gives SORTPROG one input record.

Supplying Records From SORTMERGESEND for a Single Run

You cannot supply records from both SORTMERGESEND and disk files for the same sort or merge run.

SORTMERGESEND can send input records for sorting or for merging but cannot send records for both operations in the same run.

Sending Input Records From Extended Memory

You can send each input record from an extended data segment (which must be in use at the time of the call). If you want SORTMERGESEND to send a record from a location in an extended segment, specify *record-loc-ext* instead of *record-loc* for the address of the record.

Blocking Records to Reduce Interprocess Messages

You can specify a single or double buffer for record blocking and deblocking in a call to SORTBUILDPARM. SORTMERGESTART provides the buffer to transfer a block of records instead of a single record in each interprocess message to or from SORTPROG. Each call to SORTMERGESEND puts a record into this buffer.

For more information, see SORTBUILDPARM Procedure on page 5-2.

Record blocking is valid only for sort runs. If you try to use record blocking with merge runs, SORTPROG returns sort error 81 (BLOCKED INTERFACE NOT ALLOWED WITH MERGE).

Merging Records From Input Streams

An input stream is a source of sorted records for merging. You can specify up to 32 input streams in the call to SORTMERGESTART, with the *num-merge-files* and *in-file-name* parameters. The *in-file-name* parameter must specify all blanks as the name for each input stream.

The first call to SORTMERGESEND sends the first input record from stream 0. After each call, SORTMERGESEND puts the number of the next input stream that SORTPROG wants a record from in *stream-id*. Stream numbers are consecutive integers.

When an input stream has no more records, you set the *length* parameter of SORTMERGESEND to –1. When all input streams have no more records, SORTMERGESEND sets *stream-id* to –1; then SORTPROG finishes merging the records and produces the output file or returns the records through SORTMERGERECEIVE.

Examples

In this example, INBUF contains one input record, and INLEN is the number of bytes in the record:

```
send^status := SORTMERGESEND (sortblock,
                              inbuf,
                              inlen);
```

In the next example, INBUF^EXT is the extended memory location of an input record. Commas reserve places for the *record-loc*, *stream-id*, *spare1*, and *spare2* parameters:

```
send^status := SORTMERGESEND (sortblock,,
                              inlen,,,,
                              inbuf^ext);
```

SORTMERGESTART Procedure

Use SORTMERGESTART to start the SORTPROG process and pass parameters to SORTPROG for a sort or merge run. This procedure begins every run when you use

FastSort through a program. A COBOL85 program can call SORTMERGESTART through the SORT or MERGE statement.

```
{ status := } SORTMERGESTART ( ctlblock                ! i
{ CALL      }          ,key-block                      ! i
                       ,[ num-merge-files ]            ! i
                       ,[ num-sort-files ]             ! i
                       ,[ in-file-name ]               ! i
                       ,[ in-file-exclusion-mode ]     ! i
                       ,[ in-file-count ]              ! i
                       ,[ in-file-record-length ]      ! i
                       ,[ format ]                     ! i
                       ,[ out-file-name ]              ! i
                       ,[ out-file-exclusion-mode ]    ! i
                       ,[ out-file-type ]              ! i
                       ,[ flags ]                      ! i
                       ,[ errnum ]                     ! o
                       ,[ errproc ]                    ! i
                       ,[ scratch-file-name ]          ! i
                       ,[ scratch-block ]              ! i
                       ,[ process-start ]              ! i
                       ,[ max-record-length ]          ! o
                       ,[ collate-sequence-table ]     ! i
                       ,[ dslack ]                     ! i
                       ,[ islack ]                     ! i
                       ,[ flags2 ]                     ! i
                       ,[ subsort-count ]              ! i
                       ,[ spare5 ]  )                  !
reserved
```

status returned value

INT

returns a FastSort error code if an error occurred; if not, returns 0.

ctlblock input

INT:ref:200

is a 200-word integer array that FastSort procedures use as an internal control block to store information. After the calling process declares *ctlblock*, it must not alter any values in the control block between the call to SORTMERGESTART and the call to SORTMERGEFINISH; otherwise, the SORTPROG process returns a FastSort error code and stops. Also, do not rely on the information in *ctlblock*, because this information can change without warning.

`key-block` `input`

INT:ref:*

is an integer array defining the key fields. Its size is one word plus three words for each key. The first word contains the total number of keys. The rest of the array contains three-word descriptions of the keys. The maximum number of keys is 63.

For more information, see Key-Field Definitions in the Key-Block Array on page 5-30."

`num-merge-files` `input`

INT:value

is the number of input files for merging. The amount of space available for the scratch file determines the maximum number of records SORTPROG accepts. The total number of files for both sorting and merging must be greater than 0 and cannot exceed 32. If you omit the *`num-sort-files`* parameter (or specify a value of 0), you must specify a value for *`num-merge-files`*.

If the merge files are input streams, the *`in-file-name`* parameter must specify all blanks as the name of each stream, and SORTMERGESEND must send each record to SORTPROG. For more information, see Merging Records From Input Streams on page 5-19.

`num-sort-files` `input`

INT:value

is the number of input files for sorting. The amount of space available for the scratch file determines the maximum number of records SORTPROG accepts. The total number of files for both sorting and merging cannot exceed 32. If you omit the *`in-file-name`* parameter, the number of sort files must be 1. If you omit the *`num-merge-files`* parameter (or specify a value of 0), you must specify a value for *`num-sort-files`*.

`in-file-name` `input`

INT:ref:*

is an array including one 12-word entry for each input file. SORTPROG accepts a set of input files in the order presented, with the merged files first. If you specify more than one input file, you must specify a name for each file. The name can be all blanks for a single sort file. Each name must be all blanks if the input files are streams for merging.

When working with more than one input file, SORTPROG uses the same key-field specifications for all input records.

You can specify files containing sorted records and files containing unsorted records for the same sort run.

If you omit *`in-file-name`* or if it equals all blanks, your process must call SORTMERGESEND to send each input record to SORTPROG. SORTMERGESEND cannot send both sorted and unsorted records for the same sort run. For more information, see SORTMERGESEND Procedure on page 5-15.

You can specify the same file in both *`in-file-name`* and *`out-file-name`* for a sort run but not for a merge run.

△ **Caution.** If you specify the same file as both an input file and the output file for a sort run, you can lose all the data from the input file if an error or processor failure terminates the SORTPROG process.

`in-file-exclusion-mode` `input`

INT:ref:*

is an array including a one-word entry for each input file. Each entry contains the exclusion mode SORTPROG uses when it opens the corresponding input file. If you specify an exclusion mode for one input file, you must specify a mode for each input file for a sort or merge run. Use one of the following values to specify the exclusion mode:

Value	**Exclusion Mode**
–1	Use the default mode
0	SHARED
1	EXCLUSIVE
2	PROTECTED

For SHARED access, if another process is writing to the input file while FastSort is reading it, the operating system might return file-system error 59 (FILE IS BAD). However, the file is not necessarily corrupted. Retry the sort or merge run.

If you specify PROTECTED for the *`in-file-exclusion-mode`* parameter, the *`in-file-name`* parameter cannot specify the same file name as the *`out-file-name`* parameter; otherwise, SORTPROG returns sort error 49 (INVALID EXCLUSION MODE SPECIFIED).

If you specify –1 or omit this parameter, these default exclusion modes apply:

Device	**Exclusion Mode**
Permanent disk file	PROTECTED
Temporary disk file	SHARED
Terminal	SHARED
Other	EXCLUSIVE

If you want your process to read the input file at the same time as SORTPROG, specify PROTECTED exclusion mode for SORTPROG and use SHARED exclusion mode when your process opens the file.

in-file-count `input`

INT(32):ref:*

is an array including one 32-bit entry for each input file. Each entry contains the maximum number of records in the corresponding input file. When input is from a source other than disk, SORTPROG uses *in-file-count* to estimate the space required for the scratch file.

If you omit *in-file-count* or specify –1, SORTPROG determines the maximum number of records as follows:

- For a structured disk file, SORTPROG estimates the number of records in the file by looking at the end-of-file location and determining the structured overhead.
- For an unstructured disk file, SORTPROG calculates an approximate number of records in the file. The approximate number of records for an EDIT file is the end-of-file location multiplied by 2 and divided by the record length. The approximate number of records for other unstructured files is the end-of-file location divided by the record length. The default record length for unstructured files is 132 bytes.
- For files other than disk files and records supplied by SORTMERGESEND, the default number of records is 50,000.

in-file-record-length `input`

INT:ref:*

is an array including one 16-bit entry for each input file. Each entry contains the maximum record length in the corresponding input file. The largest record length allowed is 4080 bytes. If you omit *in-file-length* or specify –1, SORTPROG uses the default record length.

You can omit this parameter when the input file is a structured disk file, because the length is in the file label.

To use an odd unstructured file for an input file, you must specify the correct length in *in-file-record-length*. For unstructured disk files, files other than disk files, and records supplied by SORTMERGESEND, the default maximum record length is 132 bytes.

format `input`

INT:value

specifies the output record format with one of these values:

0	The output records are in the same format as the input records. This is a record sort, the default SORTPROG uses when you omit *`format`*.
1	The output records are 32-bit integers describing the order of the sorted records. This is a permutation sort. For example, if the 20th input record is first in order after sorting, 20 is the value of the first output record.
2	Each output record consists of the key-field values concatenated in the order you defined the fields. This is a key sort. If a key field extends beyond the end of a variable-length record, SORTPROG pads the key values with blanks for a structured file.
3	Each output record begins with the 32-bit (4-byte) record number followed by the concatenated values of the key fields. This is a combined permutation and key sort.

`out-file-name` `input`

INT:ref:12

is a 12-word array that names the file for the output records. If you omit *`out-file-name`* or it equals all blanks, SORTMERGERECEIVE must return the records, one for each call, to the calling process.

If *`out-file-name`* specifies an existing file that has a different type than *`out-file-type`* or the default for *`out-file-type`*, SORTPROG purges the file and creates a new one with the same name.

△ **Caution.** If you specify the same file as both an input file and the output file for a sort run, you can lose all the data from the input file if an error or processor failure terminates the SORTPROG process.

`out-file-exclusion-mode` `input`

INT:value

is the exclusion mode with which SORTPROG opens the output file. Use one of the following values to specify the exclusion mode:

Value	Exclusion Mode
–1	Use the default mode
0	SHARED
1	EXCLUSIVE
2	PROTECTED

If you specify –1 or omit this parameter, FastSort uses one of the following default exclusion modes:

Device	**Exclusion Mode**
Disk or magnetic tape file	EXCLUSIVE
Temporary disk file	SHARED
Terminal	SHARED

out-file-type `input`

INT:value

specifies the type of file SORTPROG creates for the output records. Use one of the following codes to specify a file type:

Code	**File Type**
–1	Default (same effect as omitting parameter)
0	Unstructured file
1	Relative file
2	Entry-sequenced file
3	Key-sequenced file

You can omit this parameter if you omit *out-file-name* or if *out-file-name* equals all blanks. In this case, SORTMERGERECEIVE returns the output records to the calling process. For more information, see SORTMERGERECEIVE Procedure on page 5-13.

The default for *out-file-type* is the file type of the existing output file, if any, or of the first input file. SORTPROG can send output to key-sequenced files but not to EDIT files.

To use an odd unstructured file for *out-file-name*, create the file using the FUP CREATE command or the CREATE system procedure before the sort or merge run. Then set *out-file-type* to –1.

flags `input`

INT:value

directs SORTPROG to perform a specific set of operations as shown in Table 5-4 on page 5-32. If you set *flags*.<15> to 1 (the restart option) and change the current *process-start* parameters, an existing SORTPROG process ignores the changes, except for the priority word. Set the unused *flags* bits to 0.

To use nowait I/O, specify the *flags2* parameter.

errnum `output`

INT(32):ref:1

receives a completion code of 0 if no error occurred or receives error codes if an error occurred. The high-order word has the file-system or NEWPROCESS error code. The low-order byte of the low-order word has the FastSort error code:

Parameter Word	Bits															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
High-Order	**File-System or NEWPROCESS[NOWAIT] Error Code**															
Low-Order	**FastSort Input File Index**							**FastSort Error Code**								

VST503.vsd

You can use SORTERRORSUM to supply the text of the FastSort error message and to get the index of the input file in the `in-file-name` array. SORTERRORSUM also identifies the output file, scratch file, or SORTPROG process that caused an error.

Note. FastSort saves the address of *`errnum`* in its control block. If an error occurs for calls to SORTMERGESEND, SORTMERGERECEIVE, SORTMERGEFINISH, SORTMERGESTATISTICS, or SORTMERGESTART, FastSort returns an error to this address. A user procedure that calls these procedures can access *`errnum`*. However, HP recommends that you call SORTERRORDETAIL rather than use the *`errnum`* parameter to get error information.

`errproc` `input`

is a procedure that FastSort can call when an error occurs. For more information, see Writing a User Error Procedure on page 5-37.

`scratch-file-name` `input`

INT:ref:12
INT:ref:*

is a 12-word name for an initial scratch file or an array with a file name for each subsort scratch file. You can name only a scratch volume in the first 8 bytes and leave the remaining bytes blank. If you specify an existing file, it must be unstructured.

If you specify *`subsort-count`*, the size of the *`scratch-file-name`* array is the value of *`subsort-count`* + 1. The first file name is a scratch file for the distributor-collector process, and each additional file name is an initial scratch file for a subsort process, as in this example:

```
INT DIST^SCRATCH[0:11] := ["                        "];
INT SUB1^SCRATCH[0:11] := ["$DATA1                  "];
INT SUB2^SCRATCH[0:11] := ["$DATA2                  "];
```

If you do not specify *`subsort-count`*, then *`scratch-file-name`* is the initial scratch file for a serial sort or merge run. If you omit *`scratch-file-name`* or it equals all blanks, SORTPROG creates a scratch file on a suitable volume. You cannot omit or use blanks for a subsort *`scratch-file-name`*. For more information about scracth files, see Table 5-4 on page 5-32.

`scratch-block` `input`

INT:value

is the size, in bytes, of input and output blocks for SORTPROG scratch files. The *`scratch-block`* value can be any multiple of 2048 up to 56 KB. The value must be large enough to accept the largest input record, rounded up to the nearest even byte, plus 14 bytes of overhead.

If you omit *`scratch-block`* or specify –1, the default value for *`scratch-block`*, SORTPROG uses the default scratch block size of 56 KB.

For parallel sorting, specify *`scratch-block`* only for the distributor-collector process. The subsort processes use the same block size as the distributor-collector.

`process-start` `input`

INT:ref:*

specifies the parameters for starting each SORTPROG process.

For serial sorting, *`process-start`* uses only the first four words of the NEWPROCESS structure.

For parallel sorting, *`process-start`* must use the 29-word expanded NEWPROCESS structure and include an entry for each process, starting with the distributor-collector. You must set *`flags`*.<6> to 1 to use the 29-word expanded NEWPROCESS structure. Table 5-2 on page 5-28 describes the layout of this structure.

Table 5-2. Expanded NEWPROCESS Structure

Word	Entity	Description
0*	Priority	Assigns the priority of the SORTPROG process. If priority equals –1, the default value, the SORTPROG process has the same priority as the calling process.
1*	Memory	Specifies the maximum number of data pages the SORTPROG process can use. SORTPROG always uses 64 for this value.
2*	CPU	Specifies the number of the processor (CPU) in which SORTPROG runs. If CPU equals –1, the default value, SORTPROG runs in the same processor as the calling process or, in parallel sorting, in a processor that FastSort selects. You can specify a group of processors for FastSort to select from by using the *cpu-mask* or *not-cpu-mask* parameters or both in a call to SORTBUILDPARM.
3*	System	Specifies the number of the system in which SORTPROG runs. If system equals –1, the default value, SORTPROG runs on the same node as the calling process. See the LOCATESYSTEM procedure in the *Guardian Procedure Calls Reference Manual*. FastSort does not use this parameter for parallel sort runs.
4	Segment	Specifies the size of the extended memory segment from 256 to 62,255 pages. If segment equals –1, the default value, segment size is controlled by the MINTIME and MINSPACE flags of the `flags` parameter. The size cannot be more than 90 percent of the processor’s physical memory not locked down by the operating system. To use this parameter, you must set *flags*.<6> to 1. The segment size you specify overrides the MINSPACE or MINTIME flag. To specify a segment size of greater than 32,767 you must set `flags2`.<4> to 1.
5:16	Swap-file [0:11]	Specifies the name of the swap file for the extended memory segment. The swap file must be on the local node. If swap-file equals all blanks, the default value, FastSort creates a temporary file on the same volume as the scratch file if the scratch file is local. If not, FastSort creates a temporary file on the volume where SORTPROG is running. To use this parameter, you must set *flags*.<6> to 1.
17:28	Sort-program [0:11]	Specifies the name of a file that contains the SORTPROG program. If sort-program equals all blanks, the default value, FastSort runs the program in $SYSTEM.SYS*nn*.SORTPROG. To use this parameter, you must set *flags*.<6>to 1.

***** The `process-start` structure consists of words 0-3 of this NEWPROCESS structure.

If you use `process-start`, you must specify a value for each parameter in every 29-word entry. You can specify the default value for any parameter. The default value for swap-file and sort-program is all blanks. The default value for the other parameters is –1.

`max-record-length` `output`

INT:ref:*

You should specify *`max-record-length`* as a reference to a single 16-bit word used for OUTPUT. In *`max-record-length`*, SORTMERGESTART returns the size of the largest output record that FastSort writes to the output file or returns through SORTMERGERECEIVE.

`collate-sequence-table` `input`

STRING:ref:256

is a 256-byte array defining an alternate collating sequence for SORTPROG to use in the sort or merge run. This parameter applies to alphanumeric string items only. SORTPROG uses each alphanumeric character as an index into the collating table to obtain the value to use for comparisons.

To cause SORTPROG to use the alternate collating sequence table, you also need to set *`flags`*.<10> to 1. If this bit is 0, SORTPROG ignores *`collate-sequence-table`*.

`dslack` `input`

INT:value

specifies the percentage of data slack for a key-sequenced output file. The range is 0 - 99 and the default is 0.

`islack` `input`

INT:value

specifies the percentage of index slack for a key-sequenced output file. The default is 0 slack.

`flags2` `input`

INT:value

directs FastSort to use nowait I/O if *`flags2`*.<15> is set to 1. To use nowait I/O, you must also specify the *`buffer2`* parameter in the SORTBUILDPARAM procedure call; otherwise FastSort returns sort error 74 (INVALID BLOCK ADDRESS SPECIFIED).

If you specify nowait I/O, the FastSort routines call AWAITIO when necessary. Your process should not call AWAITIO to wait on any file (*`filenum`* = –1). For more information about AWAITIO, see the *Guardian Procedure Calls Reference Manual*. For information about nowait I/O, see the *Guardian Programmer's Guide*.

Also directs FastSort to use up to 127.5 MB of extended memory if available and *`flags2.`*<4> is set to 1. Note that if you set *`flags2.`*<4> to 0, FastSort does not turn VLM off. For more information about VLM, see Using VLM on page 9-10.

Other *flags2* bits are not used and should be set to 0.

subsort-count `input`

INT:value

specifies the number of subsort processes from 2 to 8. Higher numbers can cause run-time errors, depending on your system configuration and the system load. If *subsort-count* equals *n*, *scratch-file-name* and *process-start* become integer arrays of dimension *n* + 1.

spare5 `reserved`

INT:value

is reserved for a future parameter. If you specify a value for *spare5*, FastSort returns an error.

Guidelines

Follow the guidelines on the next pages when you call the SORTMERGESTART procedure.

Data Stack Space

In addition to the 350 words required by system procedure calls, the SORTMERGESTART procedure requires additional data stack space that is not automatically allocated by the BINSERV process during compilation. Refer to the table at the beginning of this section as a guideline to determine the amount of additional space you need to allocate for an application that calls SORTMERGESTART.

Key-Field Definitions in the Key-Block Array

The first word of the key-block array is the number of keys. After the first word, each three words describe a key as follows:

Word	**Description**
0	Number of Keys
1:3	Description of First Key
4:6	Description of Second Key
	. . .
	Description of Last Key

Table 5-3 on page 5-31 lists the values for each three-word key description.

Table 5-3. Key-Field Definitions

Word	Bit Positions	Values and Description	
0	<0>	O	0 = Ascending order
			1 = Descending order
	<1>	U	0 = Do not upshift
			1 = Upshift (alphanumeric string only)
	<2:7>	R	Reserved; must be 0
	<8:15>	Type	1 = ALPHANUMERIC STRING
			2 = UNSIGNED NUMERIC STRING
			3 = NUMERIC STRING SIGN, TRAILING EMBEDDED
			4 = NUMERIC STRING SIGN, TRAILING SEPARATE
			5 = NUMERIC STRING SIGN, LEADING EMBEDDED
			6 = NUMERIC STRING SIGN, LEADING SEPARATE
			9 = BINARY SIGNED
			10 = BINARY UNSIGNED
			11 = FLOAT
1	<0:15>	Length	Key length in bytes. For key type 11, length must be 4 or 8 bytes. For key types 3 through 6, length must be 32 or fewer bytes.
2	<0:15>	Offset	Offset from beginning of record to key in bytes (record begins at 0). For key type 11, offset must be an even number.

If a key field extends beyond the end of a variable-length record, SORTPROG pads the concatenated key values with blanks in a structured output file. SORTPROG can compare an alphanumeric key field at the end of a short record if the record contains the first byte of the key value.

Fields of the flags Parameter

Table 5-4 on page 5-32 lists the *`flags`* parameter bits for the SORTMERGESTART procedure.

Table 5-4. SORTMERGESTART *flags* Parameter Bits (page 1 of 2)

Flag Meaning	*flags* Bit	Value	Description
No purge of existing output file	<5>	0	SORTPROG purges an existing output file that seems too small. This value is the default.
		1	SORTPROG does not purge an existing output file that seems too small, unless the file has the wrong file type or maximum record length.
Structure for NEWPROCESS parameters	<6>	0	SORTPROG uses the *process-start* four-word structure described in Table 5-2 on page 5-28. This value is the default.
		1	SORTPROG uses the expanded *process-start* structure (an array of one or more 29-word entries) shown in Table 5-2 on page 5-28. This value is required for parallel sort or merge runs.
MINSPACE mode	<7>	0	SORTPROG does not use MINSPACE mode. If the MINTIME flag is set to 0 and *process-start* does not specify a segment size, SORTPROG uses the AUTOMATIC mode, in which the extended memory segment size is limited to 50% (90% for the distributor-collector in parallel sorting) of the available physical memory. This value is the default.
		1	SORTPROG uses MINSPACE mode, in which the extended memory segment size is 256 pages.
MINTIME mode	<8>	0	SORTPROG does not use MINTIME mode. If the MINSPACE flag is set to 0 and `process-start` does not specify a segment size, SORTPROG uses the AUTOMATIC mode, in which the extended memory segment size is limited to 50% (90% for the distributor-collector in parallel sorting) of the available physical memory, up to 127.5 MB. The default is 56 MB.
		1	SORTPROG uses MINTIME mode, in which the extended memory segment size is limited to 70 percent of the available physical memory.
Scratch file size check	<9>	0	Exists only for compatibility with earlier versions of FastSort.
		1	Exists only for compatibility with earlier versions of FastSort.
Alternate collating sequence table	<10>	0	SORTPROG ignores the *collate-sequence-table* parameter. This value is the default.
		1	SORTPROG uses the alternate collating sequence table. If you did not provide the table, SORTMERGESTART returns error 67.

Table 5-4. SORTMERGESTART *`flags`* Parameter Bits (page 2 of 2)

Flag Meaning	*flags* Bit	Value	Description
Removal of records that have duplicate keys	<11>	0	SORTPROG keeps all records that have duplicate keys. This value is the default.
		1	SORTPROG removes every record whose keys are all duplicates of a previous record's keys.
Saving scratch files	<12>	0	SORTPROG purges scratch files after the sort run. This value is the default.
		1	SORTPROG saves a permanent scratch file if you named it.
Creating a new scratch file	<13>	0	If a scratch file exists, SORTPROG purges all data from the file and uses it. This value is the default.
		1	If a scratch file exists, SORTPROG purges it and creates a new one, unless the existing file is a temporary file created by your process.
Creating a new output file	<14>	0	If the output file exists, is large enough to hold the output, and has the specified file type and maximum record length, SORTPROG purges all data from the file and uses the file. This value is the default.
		1	If the output file exists, SORTPROG purges it and creates a new one, unless the existing file is a temporary file created by your process.
Restart option	<15>	0	SORTMERGESTART starts a new SORTPROG process. This value is the default.
		1	SORTMERGESTART uses the existing SORTPROG process and does not start a new one. If SORTPROG stops, it uses the current parameters upon restarting. If SORTPROG exists and the current *process-start* parameters are different from when it started, SORTPROG ignores all changes except a changed priority value.

Input Files

Follow these guidelines for input files:

- SORTPROG accepts all types of input files except blocked tape files and processes.
- SORTPROG accepts up to 32 input files. The files can contain fixed-length or variable-length records.
- The sum of *`number-merge-files`* and *`number-sort-files`* must be at least 1 file. Although both parameters are optional, you must specify one of them.
- SORTPROG cannot accept input records from blocked tape files. Before presenting these files to SORTPROG, use the File Utility Program (FUP) to deblock the records. For information about FUP, see the *File Utility Program (FUP) Reference Manual*.
- SORTPROG cannot accept records greater than 4072 bytes directly from input files, SORTMERGESEND or SORTMERGERECEIVE must be used to send or receive these records.

Output File Types

If *`out-file-name`* specifies a nonexistent disk file or if an existing output file has the wrong maximum record length, file type, or size, SORTPROG creates a new output file. You can use the NOPURGE option (that is, set *`flags`*.<5> to 1) to tell SORTPROG not to purge an output file that seems too small. SORTPROG creates a new output file according to the following rules, in order:

1. SORTPROG uses the file type specified in the *`out-file-type`* parameter.
2. SORTPROG uses the existing *`out-file-name`* file type if it is a valid output file type.
3. SORTPROG uses the first *`in-file-name`* file type if it is a valid disk file type for output.
4. If none of the above conditions exist, SORTPROG creates an entry-sequenced file.
5. SORTPROG does not send output to EDIT files.
6. SORTPROG cannot write records with length greater than 4072 bytes directly to the output file.

You can use a process as an output file.

If *`out-file-name`* is a blocked tape file, SORTPROG writes one record for each block. You can use FUP to block the records and load the tape file. For information about FUP, see the *File Utility Program (FUP) Reference Manual*.

The output file type can be key-sequenced. For key-sequenced files, these rules apply:

- You can use only one sort key field, and the data type for the field must be BINARY UNSIGNED.
- The sort key field must be the same as the file's primary key field.
- You must specify ascending in the *key-block* array.

You can specify the *dslack* and *islack* parameters for an existing key-sequenced output file.

Existing Output Files

If *out-file-name* exists on a disk prior to the sort or merge run, SORTPROG purges all the data in the file before reusing it. For SORTPROG to reuse an existing disk file as an output file, all of the following cases must be true:

- The existing file type must be the same as the output file type in effect for the run.
- The existing file size must be equal to or greater than the sum of the all the input file sizes, except when you specify the NOPURGE option (*flags*.<5> set to 1).
- The maximum record length for the existing file must be equal to or greater than the maximum output record length for the run.

If any of these is not true, SORTPROG purges the existing output file and creates a new file. If you do not want FastSort to purge and recreate the file, set *flags*.<5> to 1.

△ **Caution.** If you specify the same file as both an input file and output file for a sort run, you can lose all the data from the input file if an error or processor failure terminates the SORTPROG process.

Record Count

The value of *in-file-count* need not be the exact number of records in the input file. However, you should round up and not down if you round off the number of records.

If you underestimate the number of input records, SORTPROG might underestimate the size needed for the scratch or output file, which can cause FastSort error 29 or 30. For more information on error messages, see Appendix B, FastSort Error Messages.

Extended Memory Size

For more information, see Controlling Extended Memory on page 2-11.

Restart Option

This option enables the same SORTPROG process to be used for successive sort or merge runs. The requirements for using the restart option are:

- Set the restart flag (*flags*.<15>) to 0 for the first call to SORTMERGESTART and to 1 for successive calls.

- Before your process can call SORTMERGESTART with the restart flag set to 1, your process must call SORTMERGESTATISTICS or an error must end the SORTPROG process.
- Each call to SORTMERGESTART must specify the same sort control block.
- If you call the SORTBUILDPARM procedure, its restart flag (*build-flags*.<15>) must be set to 0 for the first call and to 1 for successive calls.
- Each call to SORTBUILDPARM must specify the same sort control block as the call to SORTMERGESTART.

A call to SORTMERGESTART returns immediately after SORTPROG reads the input parameters. When your process calls SORTMERGESTART again, SORTPROG accepts parameters for restart as follows:

- If SORTPROG ends abnormally and the restart flag is set to 1, SORTPROG uses the most recently specified *process-start* parameters when it restarts.
- If SORTPROG exists and the current *process-start* parameters are different than when it started, SORTPROG ignores all changes except for a changed priority value.

The COBOL85 SORT and MERGE statements do not support the restart option.

Alternate Collating Sequence Table

The calling process can read an alternate collating sequence table from a file that the COLLATEOUT command produced.

Example

The following example shows the SORTMERGESTART procedure with the restart option:

```
error := SORTMERGESTART (sortblock,
                         keyblock,,
                         sortfiles,,,,
                         len,,,,,
                         restart); ! restart.<15> is 0

  ...

! Call SORTMERGESEND for each record
! Call SORTMERGESEND with length = -1
! Call SORTMERGERECEIVE for each record, until length = -1

errnum := SORTMERGESTATISTICS (sortblock,
                               statlen,
                               stats);
! statlen = 21, and stats is an integer array of 21 words.
  ...
error := SORTMERGESTART (sortblock,
                         keyblock,,
                              sortfiles,,,,
                              len,,,,,
                              restart);  ! restart.<15> is 1

! Go to the first statement that calls SORTMERGESEND.
  ...
error := CALL SORTMERGEFINISH (sortblock);
  ...
END;            ! End of the routine
```

Writing a User Error Procedure

You can use the *errproc* parameter of SORTMERGESTART to specify a TAL procedure to call if an error ends the SORTPROG process.

```
PROC errproc ( code )
```

errproc

is the name of the user error procedure that you specify in the *errproc* parameter of SORTMERGESTART.

code

`INT(32):value`

returns error codes to *errnum*, which you specify in the call to SORTMERGESTART. Both *code* and *errnum* have the same structure.

Shown below is the TAL syntax for the declaration of a user error procedure. The body of the user error procedure contains TAL declarations and statements. For information about TAL, see the *TAL Reference Manual*.

```
PROC sorterrproc (errcode);
  INT(32) errcode;
  BEGIN
 ...     .      ! TAL statements
  END;
 ...
  error := SORTMERGESTART ( sortblock , keys
                      ,            ! num-merge-files
                      ,numsort
                      ,infile
                      ,            ! in-file-exclusion-mode
                      ,            ! in-file-count
                      ,            ! in-file-record-length
                      ,            ! format
                      ,outfile
                      ,            ! out-file-exclusion-mode
                      ,            ! out-file-type
                      ,            ! flags
                      ,            ! errnum
```

```
                       ,sorterrproc );
   ...
```

SORTMERGESTATISTICS Procedure

Use SORTMERGESTATISTICS to obtain information about a successful sort or merge run after SORTPROG completes the run.

```
{ status := } SORTMERGESTATISTICS ( ctlblock      ! i
{ CALL      }                      ,length        ! i, o
                                   ,statistics    ! o
                                   ,[flag1 ]      ! i
                                   ,[spare1 ]     !reserved
```

status returned value

returns a FastSort error code if an error occurred; if not, *status* returns 0.

ctlblock input

INT:ref:200

is the global storage array named in the call to SORTMERGESTART. You should not rely on the information in *ctlblock*, because this information can change without warning.

length input, output

INT:ref:*

indicates the length, in words, of the SORTPROG statistics that SORTMERGESTATISTICS returns after run completion. You can set *length* to the number of words you want returned, from 1 to 22. When SORTMERGESTATISTICS returns statistics, it sets *length* to the number of words actually returned. The default value for *length* is 0, which causes SORTMERGESTATISTICS to not return any statistics.

Values less than 0 or greater than 21 when *flag1* = 0 or greater than 22 when *flag1* = 1 will yield error 149 (INVALID STATISTICS LENGTH SPECIFIED).

statistics output

INT:ref:21 (*flag1* = 0 or does not exist)
INT:ref:22 (*flag1* = 1)

is a 21-word or 22-word array into which SORTPROG returns the statistics. The array is 21 words long if VLM is off and 22 words long if VLM is on. For a description of this array, see Table 5-5 on page 5-39. For more information about the VLM option, see Using VLM on page 9-10.

`flag1` input

INT:value

tells FastSort to use the 22-word array to return statistics if this parameter is present and set to 1. If `flag1` is present but set to 0 or if `flag1` is not present, FastSort uses the 21-word statistics array and converts BUFFER PAGES from an INT(32) to an INT value before it reaches the array. For BUFFER PAGES, FastSort returns the value -1 for values greater than 32,767.

Values other than 0 or 1 for `flag1` yield error 69 (INVALID STATISTICS FLAG SPECIFIED).

`spare1` reserved

is reserved for future parameters. If you specify a value for `spare1`, FastSort returns an error.

The SORTMERGESTATISTICS *statistics* Structure

SORTMERGESTATISTICS *statistics* is a 21-word array when the VLM option is off and a 22-word array when VLM is on. For more information about this option, see Using VLM on page 9-10.

Table 5-5 on page 5-39 describes the SORTMERGESTATISTICS *statistics* structure. The uppercase terms show the equivalent statistics that FastSort returns after an interactive run.

Table 5-5. SORTMERGESTATISTICS *statistics* Structure

Word	Type	Description
0	INT	MAX RECORD SIZE: maximum record size in bytes
1	INT or INT(32)*	BUFFER PAGES: number of 1,024-word pages of extended memory SORTPROG used as a sort area.
2:3	INT(32)	RECORDS: number of records
4:5	INT(32)	ELAPSED TIME: total time SORTPROG took to process the sort or merge request to the nearest hundredth of a second
6:7	INT(32)	COMPARES: number of times SORTPROG compared two records
8:9	INT(32)	SCRATCH SEEKS: number of blocked read and write operations on the scratch file
10:11	INT(32)	I/O WAIT TIME: the time SORTPROG spent on calls to READ, WRITE, and AWAITIO, to the nearest hundredth of a second
12:13	INT(32)	SCRATCH DISK: number of bytes in the scratch file
14:15	INT(32)	INITIAL RUNS: number of runs generated by the first pass
16	INT	FIRST MERGE ORDER: number of runs merged in the first intermediate pass

***** If VLM is on, BUFFER PAGES is an INT(32) value and all subsequent words in this array move up one word.

Table 5-5. SORTMERGESTATISTICS *`statistics`* Structure

Word	Type	Description
17	INT	MERGE ORDER: maximum number of runs that can be merged at one time
18	INT	INTERMEDIATE PASSES: number of merge cycles between initial run formation and final merge
19:20	INT(32)	NUMBER OF DUPLICATES: number of duplicate records SORTPROG removed

***** If VLM is on, BUFFER PAGES is an INT(32) value and all subsequent words in this array move up one word.

Example

```
stat^error := SORTMERGESTATISTICS (sortblock,
                                   length,
                                   statistics);
```

6 Sorting in Parallel

If the total input file size is larger than one megabyte, a parallel sort run can provide better performance in elapsed execution time than a serial sort run. A parallel sort operation improves performance because it:

- Distributes the workload to multiple processors
- Uses scratch files on multiple disks

For a parallel sort run, you set up a distributor-collector process and from 2 to 8 subsort processes.

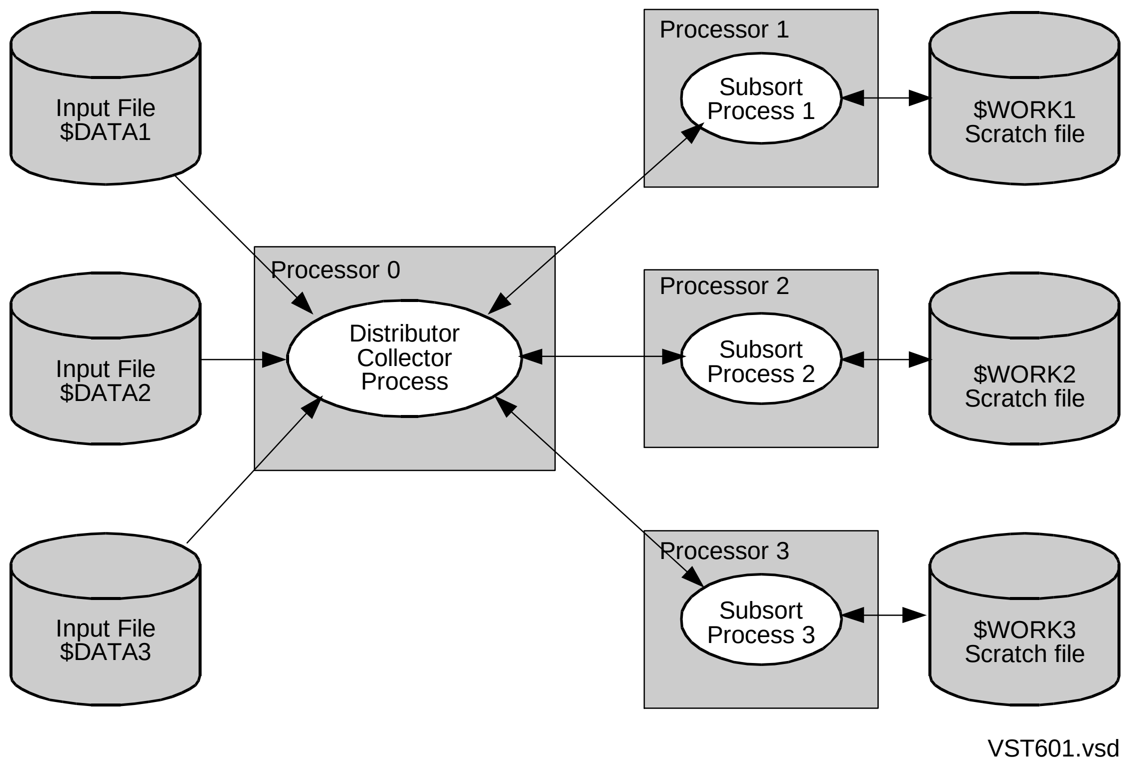
Note. Although you can specify a maximum of 16 subsort processes, HP recommends you specify no more than 8 processes. More than 8 subsort processes can cause a parallel sort run to fail with FastSort error 22 (THE MEMORY SPACE FOR SORTING IS INSUFFICIENT).

The distributor-collector and subsort processes are SORTPROG processes. You can use either FastSort commands or system procedures to set up the distributor-collector and subsort processes. You run each subsort process in a different processor and assign each processor a different disk for scratch files.

The distributor-collector process reads input files and distributes the records among the subsort processes. Each subsort process sorts its portion of the records. The distributor-collector process then collects the sorted records, merges them, and produces the output file.

Figure 6-1 on page 6-2 shows how multiple SORTPROG processes work together in a parallel sort run.

Figure 6-1. Parallel Sorting



This section gives guidelines for using FastSort commands, procedures, and parameters for parallel sorting. For more information about commands, see Section 3, Using FastSort Commands and for more information about procedures, see Section 5, Using FastSort System Procedures. For information about using partitioned input and output files, see Partitioned Files on page C-5.

Using Commands for Parallel Sorting

To use FastSort commands to set up a parallel sort run, follow these steps:

1. Name and describe any input and output files in FROM and TO commands.
2. Define your key fields for sorting in one or more ASCENDING or DESCENDING commands or a combination of both.
3. If you want to use an alternate collating sequence, name a file containing the sequence in the COLLATE command.
4. Set up individual subsort processes with SUBSORT commands, one command for each process. For more information, see Using the Automatic Configuration on page 6-4 and Configuring Subsort Processes on page 6-6.

5. If you want to specify a group of processors for running subsort processes, list the processors SORTPROG can use in a CPUS command. Specify any processors SORTPROG cannot use in a NOTCPUS command. For more information, see Selecting Processors to Run Subsort Processes on page 6-7.
6. Set up the distributor-collector process in the RUN command that starts the parallel sort operation. For more information, see Using the Automatic Configuration on page 6-4 and Configuring a Distributor-Collector Process on page 6-10.

The following commands start a distributor-collector process in the same processor (CPU) in which the SORT process is running and a subsort process in each of the three processors that control the disk volumes $VENUS, $MARS, and $SATURN. The volume names in the SUBSORT commands specify volumes for the temporary scratch files on $VENUS, $MARS, and $SATURN.

```
FROM LASTNAME
TO ZIPCODE
ASC 76:80, 1:15 UPPER
SUBSORT $VENUS
SUBSORT $MARS
SUBSORT $SATURN
RUN
```

FastSort tries to put each subsort process in the same processor as the primary disk process for the scratch volume. For the interactive interface, the default processor for the distributor-collector process is the same processor in which the SORT process is running.

Using Procedures for Parallel Sorting

To set up a parallel sort run by using FastSort procedures, call the procedures from your program in this order:

1. SORTBUILDPARM stores parameters for parallel sorting in the sort control block, including the numbers of the processors in which to run subsort processes. For more information, see Configuring Subsort Processes on page 6-6.
2. SORTMERGESTART begins the distributor-collector process and passes parameters to it for the sort run, including key fields for sorting, names of input and output files, and information about the subsort processes. For more information, see Configuring Subsort Processes on page 6-6 and Configuring a Distributor-Collector Process on page 6-10.
3. SORTERRORSUM returns detailed information about an error to your process, which should call this procedure only if an error occurs.
4. SORTMERGESTATISTICS ends the sort run and returns information to your process about the data sorted, the sorting and merging operations, and resource usage.

5. SORTMERGEFINISH stops the distributor-collector process.

Instead of specifying input files, you can use calls to SORTMERGESEND after the call to SORTMERGESTART. Instead of specifying an output file, you can use calls to SORTMERGERECEIVE after the last call to SORTMERGESEND, if any, or after the call to SORTMERGESTART. The TAL example in Example 6-3 on page 6-23 shows how to use procedure calls for a parallel sort run.

You can use SORT and SUBSORT DEFINEs to set up a parallel sort run. For more information, see Section 7, Using SORT and SUBSORT DEFINEs.

Using the Automatic Configuration

The simplest way to set up a parallel sort run is to let FastSort automatically configure the subsort processes for you as follows:

1. Specify the number of subsort processes.
2. Specify the name of an initial scratch file for each subsort process.
3. Start the run.

For scratch files, specify only the disk volume names. For optimum performance, use scratch-file volumes whose primary disk processes (DP2) run in different processors.

FastSort creates temporary initial scratch files on the disk volumes you specify and tries to put each subsort process in the same processor as the primary disk process for the initial scratch file.

If you do not specify a processor for the distributor-collector process, FastSort tries to put this process in a processor as follows:

- For the interactive interface, in the same processor in which the SORT process is running
- For the programmatic interface, in the same processor in which the calling process is running

The FastSort automatic configuration also includes:

- A block size of 56 KB for each subsort scratch file
- A memory size of 64 KB for the distributor-collector process and for each subsort process
- An extended memory segment for the distributor-collector process of at most 90 percent of the processor’s physical memory not locked down by the operating system

- An extended memory segment for each subsort process of at most 50 percent of the processor’s physical memory not locked down by the operating system
- A scratch file size for each subsort process equal to the output file size divided by the number of subsort processes plus 6 bytes per record for overhead

FastSort computes the sizes of the extended memory segments and scratch files for you.

Using FastSort Commands

To set up the automatic configuration with FastSort commands, use a SUBSORT command for each subsort process before you issue the RUN command:

```
SUBSORT $VENUS
SUBSORT $MARS
SUBSORT $SATURN
```

For the best performance, each scratch file should be on a separate disk volume, and the primary disk processes for the volumes should be running in different processors.

Using FastSort Procedures

To set up the automatic configuration using FastSort procedures, call the SORTMERGESTART procedure and specify these items:

- The number of subsort processes in the *subsort-count* parameter
- An initial scratch file for each process in the *scratch-file-name* array
- The *flags* parameter with bit 6 set to 1
- A *process-start* array

For example, this call to SORTMERGESTART in a TAL procedure sets up a parallel sort run with three subsort processes:

```
ERROR := SORTMERGESTART (sortblock,
                         keys,,
                         1,
                         indata,
                         ,,,,
                         outdata,,,
                         flags,,
                         scratchfiles,,
                         startparams,,,,,
                         3);
```

The SCRATCHFILES array contains an entry for a distributor-collector process scratch file and three entries for subsort initial scratch files. You must specify at least a volume name for each subsort initial scratch file:

```
INT distr^scratch[0:11] := ["                        "];
INT subp1^scratch[0:11] := ["$VENUS                  "];
INT subp2^scratch[0:11] := ["$MARS                   "];
INT subp3^scratch[0:11] := ["$SATURN                 "];
```

Because distributor-collector processes rarely use scratch files, you can omit the distributor-collector scratch file name.

Improving Performance

If the automatic configuration does not sort your records fast enough, try running the sort in the MINTIME mode. When you specify the MINTIME parameter in the RUN command or in the SORTMERGESTART procedure, the size of the extended memory segment for each subsort process can be up to 70 percent of the processor’s physical memory not locked down by the operating system.

The following subsections explain how to set parameters to configure the subsort and distributor-collector processes and to tune the configuration.

Configuring Subsort Processes

You can configure subsort processes through the SUBSORT command, through the procedures SORTMERGESTART and SORTBUILDPARM, or through SUBSORT DEFINEs. To get the best performance from subsort processes on your system, you might need to specify one or more of the following:

- A processor for each subsort process, preferably the processor that runs the primary disk process for the scratch volume
- A scratch file block size
- The size of the extended memory segment
- A location for the extended memory swap file
- A copy of the SORTPROG program on a disk volume other than $SYSTEM
- An execution priority

If you use commands to configure the parallel sort run, you can specify the parameters for each subsort process in a SUBSORT command. Some parameters of the RUN command also affect the configuration of subsort processes, as explained under Configuring a Distributor-Collector Process on page 6-10.

If you use procedure calls to configure the parallel sort run, you can specify processors in a call to the SORTBUILDPARM procedure and the other parameters for subsort processes in a call to the SORTMERGESTART procedure. Other parameters of SORTMERGESTART also affect the configuration of subsort processes, as explained under Configuring a Distributor-Collector Process on page 6-10.

Selecting Processors to Run Subsort Processes

When you name a scratch file for a subsort process, FastSort runs the process in the same processor that runs the primary disk process for the scratch file. If you see a bottleneck in a processor that is running a subsort process, however, you can specify a different processor for that process.

If a subsort process cannot run in the default processor, FastSort selects another one from a group of processors. You can specify which processors are in the group and which ones are not. When you specify a particular processor to run a subsort process and that processor is not available, FastSort does not select another processor but returns an error message.

You can specify a particular processor for a subsort process using one of the following methods, depending on whether you use commands or procedures:

- The CPU parameter of the SUBSORT command
- The *process-start* parameter of the SORTMERGESTART procedure
- The CPU attribute of a SUBSORT DEFINE

You can specify a group of processors for FastSort to select from by using any of the following:

- The CPUS and NOTCPUS commands (you can specify either or both commands)
- The *cpu-mask* and *not-cpu-mask* parameters of the SORTBUILDPARM procedure (you can specify either or both parameters)
- The CPUS and NOTCPUS attributes of a SORT DEFINE (you can specify either or both attributes)

For example, you have a system with eight processors, and you want to run four subsort processes to sort the records from a large file. To allow for peak capacity, do not load any of the processors over 60 percent. Processors number 2 and 5 are generally 50 to 60 percent busy. Processor 0 runs the distributor-collector process because it has the lightest load. FastSort can use the remaining processors, so you specify them in a CPUS command:

```
CPUS 1,3,4,6,7
```

Or you can combine the CPUS and NOTCPUS commands to specify the same group of processors:

```
CPUS ALL
NOTCPUS 0,2,5
```

ALL is the default value for CPUS, so you can use only the NOTCPUS command to specify the same group.

The *`cpu-mask`* and *`not-cpu-mask`* parameters of the SORTBUILDPARM procedure and the CPUS and NOTCPUS attributes of a SORT DEFINE have the same effects as the CPUS and NOTCPUS commands.

How FastSort Selects Processors

FastSort follows these steps to select a processor for a subsort process:

1. FastSort uses the processor you specified, if any. If that processor is not available, FastSort returns error code 76 (START OF SUBSORT PROCESS HAS FAILED).
2. If you did not specify a processor, FastSort uses the processor that runs the primary disk process for the initial scratch volume, unless the NOTCPUS command or the *`not-cpu-mask`* parameter of SORTBUILDPARM excludes that processor.
3. If the processor that controls the initial scratch volume is not available, FastSort uses any processor from the processor group. If you did not specify any processors to use or not to use, FastSort selects from a group of all processors. When FastSort selects processors for subsorts, it attempts to put each process in a different processor.
4. If FastSort cannot start the subsort process in a processor it selects, for example because the processor is down, it selects another processor from the group and tries to start the process in the new processor.

Specifying the Size of the Extended Memory Segment

If you do not specify an extended segment size for a subsort process, FastSort tries to use enough memory for the subsort to make only one merge pass. In the automatic configuration, the segment size is at most 50 percent of the processor's physical memory not locked down by the operating system.

FastSort computes the actual size of the segment as follows, using the segment size for a serial sort run (*s*) and the number of subsort processes (*n*):

$$s \times \sqrt{\frac{1}{n}}$$

This is the default extended segment size for a subsort process. To specify a different extended segment size, use one of these parameters:

- The SEGMENT parameter of the SUBSORT command
- The *`process-start`* parameter of the SORTMERGESTART procedure
- The SEGMENT attribute of a SUBSORT DEFINE

If you do not explicitly specify the segment size for a subsort process, the subsort process uses the same segment size as the distributor-collector process.

To specify the maximum segment size for all subsort processes, you can use:

- The MINTIME or MINSPACE parameter of the RUN command
- The *`process-start`* parameter of the SORTMERGESTART procedure
- The MODE attribute of a SORT DEFINE

Specifying a Location for the Swap File

The default location for the swap file in an extended memory segment is the same disk volume where the scratch file is located if the scratch file is local. For remote scratch files, the default is the volume where the program file is running. You can specify a volume for the swap file by using one of these parameters:

- The SWAP parameter of the SUBSORT command
- The *`process-start`* parameter of the SORTMERGESTART procedure
- The SWAP attribute of a SUBSORT DEFINE

Swapping, or paging, occurs only when the extended memory segment is larger than the available physical memory or when there is competition from other processes. To avoid swapping, specify less extended memory for the subsort process or move the process to a processor with more physical memory available or a lighter load

Using Multiple Copies of the SORTPROG Program

By default, each subsort process uses the SORTPROG program in the $SYSTEM.SYS*nn*.SORTPROG file. To run a subsort process from another local disk volume, follow these steps:

1. Duplicate the SORTPROG program to a file on the target local disk volume and use the FUP LICENSE command to license it.
2. Use one of these parameters to specify the location of the file:
   - The PROGRAM parameter of the SUBSORT command
   - The `process-start` parameter of the SORTMERGESTART procedure
   - The PROGRAM attribute of a SUBSORT DEFINE

Specifying an Execution Priority

The default execution priority for a subsort process is the operating system's default priority for a process. You can specify a different priority by using one of the following parameters:

- The PRI parameter of the SUBSORT command
- The `process-start` parameter of the SORTMERGESTART procedure
- The PRI attribute of a SUBSORT DEFINE

Configuring a Distributor-Collector Process

You can configure a distributor-collector process in a RUN command or in a call to the SORTMERGESTART procedure. For the best performance, follow as many of the guidelines in this subsection as possible when you configure the distributor-collector process:

- In a different processor from any of the subsort processes
- In the same processor as the primary disk process for the volume containing the output file or an input file
- In the processor that has the lightest load

You do not have to specify a scratch file or any other parameter for the distributor-collector process. The default processor is the processor of the program calling FastSort. If the default processor is heavily loaded or does not run the disk process for an input or output file's volume, you can specify another processor in the RUN command.

To improve performance for a parallel sort run on your system, you can specify one or more of the following options in the RUN command or in the SORTMERGESTART procedure:

- The size of the I/O blocks for all scratch files
- The size of the extended memory segment for the distributor-collector process and for each subsort process
- A location for the swap file for the distributor-collector process's extended memory segment
- A copy of the licensed SORTPROG program in a location other than the SYSTEM.SYS*nn*.SORTPROG file from which to run the distributor-collector process
- An execution priority for the distributor-collector process

Specifying a Scratch Block Size

The default scratch block size for each subsort process is 56 KB. The block size can be any multiple of 2048 bytes up to 56 KB. You can specify a scratch block size by using one of the following parameters:

- The BLOCK parameter of the RUN command
- The BLOCK parameter of the SUBSORT command
- The *`scratch-block`* parameter of the SORTMERGESTART procedure
- The BLOCK attribute of a SORT DEFINE

Any block size you specify applies to each subsort process. If there is a conflict between block sizes specified in the SUBSORT and RUN commands, SORTPROG uses the RUN command BLOCK parameter value.

Controlling the Size of Extended Memory Segments

Each SORTPROG process attempts to use enough memory to make only one merge pass. For the distributor-collector process, the default maximum size of the extended memory segment is 90 percent of the processor's physical memory not locked down by the operating system. For each subsort process, the default maximum size of the extended memory segment is 50 percent of the processor's physical memory not locked down by the operating system. The minimum size for each process is 256 KB.

You can use the MINTIME, MINSPACE, or SEGMENT parameter to specify a different extended segment size for the SORTPROG processes. If you specify the MINTIME parameter, FastSort uses at most 70 percent of the physical memory not locked down by the operating system. If you specify MINSPACE, the extended segment is only 256 KB. If you specify the SEGMENT parameter or the segment word, FastSort uses at most only the number of pages you specify.

The SEGMENT parameter or the segment word of the `process-start` parameter overrides AUTOMATIC, MINSPACE, or MINTIME. However, if you specify more than 90 percent of the processor's physical memory not locked down by the operating system, FastSort returns an error. For each subsort process, you can specify a different extended segment size than for the distributor-collector process by using one of the following parameters:

- The SEGMENT parameter of the SUBSORT command
- The `process-start` parameter of SORTMERGESTART
- The SEGMENT attribute of a SUBSORT DEFINE

Specifying a Location for the Swap File

The default location for the swap file in an extended memory segment is the initial scratch volume if the scratch file is local. If the scratch file is not local, the default location is the disk where the program file is running. You can specify another disk for the swap file with one of the following parameters:

- The SWAP parameter of the RUN command
- The `process-start` parameter of the SORTMERGESTART procedure
- The SWAP attribute of a SORT DEFINE

Swapping, or paging, occurs only when the extended memory segment is larger than the available physical memory, or when there is competition from other processes. To avoid swapping, specify less extended memory for the distributor-collector process or move it to a processor with more physical memory available or a lighter load.

Using Multiple Copies of the SORTPROG Program

By default, a distributor-collector process uses the SORTPROG program in the $SYSTEM.SYS*nn*.SORTPROG file. To run a distributor-collector process from another disk volume, follow these steps:

1. Duplicate the SORTPROG program to a file on the target volume and use the FUP LICENSE command to license it.
2. Use one of these parameters to specify file location:
   - The PROGRAM parameter of the RUN command
   - The `process-start` parameter of the SORTMERGESTART procedure
   - The PROGRAM attribute of a SORT DEFINE

Specifying an Execution Priority

The default execution priority for the distributor-collector process is the operating system's default priority for a process. You can use one of the following parameters to specify a different priority:

- The PRI parameter of the RUN command
- The `process-start` parameter of the SORTMERGESTART procedure
- The PRI attribute of a SORT DEFINE

Tuning and Testing a Configuration for Parallel Sorting

For a large sort run, you can tune and test a configuration for the optimum performance. To tune a configuration for a parallel sort run, follow these guidelines:

- Place scratch files on different disk volumes and on separate volumes from input and output files.
- Select nonmirrored disks for subsort scratch files, if possible. A sort run is faster with nonmirrored disks than with mirrored disks.
- Unless you know the workload of all processors, let FastSort select them. FastSort tries to put a subsort process in the same processor that is running the primary disk process for the subsort scratch file volume.
- Run the distributor-collector and each subsort process in different processors.

The most effective number and placement of subsort processes on your system depends on the number and type of processors, the processor workloads, the number and length of input records, and the type of output. To determine how many subsort processes to use and where to run them, follow these steps:

1. Start with three subsort processes and use the automatic configuration.
2. Measure the performance using the Measure program. For more information, see the *Measure Reference Manual.*
3. If the distributor-collector process is not at least 90 percent busy, add one or more subsort processes. If the distributor-collector process is 100 percent busy, you might need only two subsort processes.
4. Try to balance the subsort processes so that processors and disks have similar rates of use. If a scratch disk is much busier than other disks, consider moving the scratch file to another disk. If a subsort's processor is being used more than other processors, consider moving the subsort process to another processor.

5. Avoid intermediate merge passes for subsort processes. Use enough extended memory for each subsort process to make only one merge pass. For information about how much extended memory you need for each subsort process with different sizes of files, see Controlling Extended Memory on page 2-11.

If you do not have enough memory available in each processor, add enough subsort processes to limit the number of merge passes to one.

Understanding Statistics From Parallel Sorting

For a parallel sort run, FastSort returns some statistics that apply only to the distributor-collector process and other statistics that are totals for the distributor-collector process and all subsort processes as shown in the following table:

FastSort Process	**Statistics**
Distributor-Collector Process Only	RECORDS, BUFFER PAGES, ELAPSED TIME, INITIAL RUNS, I/O WAIT TIME, FIRST MERGE ORDER, SCRATCH DISK, MERGE ORDER, MAX RECORD SIZE, INTERMEDIATE PASSES
Distributor-Collector and Subsort Processes	COMPARES, SCRATCH SEEKS, NUMBER OF DUPLICATES

Identifying the Causes of Errors

When an error occurs during a parallel sort run, FastSort can identify the SORTPROG process in which the error occurred. FastSort can also tell you the name of the file that caused an error.

If you use interactive commands to set up the parallel sort run, FastSort sends error messages to the list file (which is usually your home terminal). If you use procedures to set up the parallel sort run, you can retrieve error information with the SORTERRORSUM procedure.

For example, if you specify a fully-qualified initial scratch file name for a parallel sort, SORTPROG returns the following error messages:

```
*** ERROR ***  A SCRATCH FILE CANNOT BE OPENED
OPERATING SYSTEM ERROR :  12
SCRATCH FILE: $DATA.SORT.SCRATCH
SORT PROCESS #2: (1,36)
```

Each line of the example is explained here:

- The first line contains the FastSort error text.
- The second line contains the file-system or NEWPROCESS error code.
- The third line specifies the type and name of the file that caused the error.
- The fourth line identifies the subsort process in which the error occurred and gives the CPU number and process identification number (PIN) for the process.

Parallel Sorting From C Programs

Example 6-1 shows a C program that calls FastSort system procedures to perform a parallel sort run.

Example 6-1. C Example of a Parallel Sort Run (page 1 of 5)

```
#pragma  sql wheneverlist
#pragma  symbols
#pragma  inspect
#pragma  runnable
#pragma  nolist
/*-------------------------------------------------------------*/
/*        FastSort Parallel Sort Run                           */
/*-------------------------------------------------------------*/
/* This program uses subsorts to sort an input file. Overflow  */
/* scratch volumes are specified in SORTBUILDPARM. Error       */
/* handling and displaying of statistics are stubbed out.     */
/*-------------------------------------------------------------*/
/* External declarations                                       */
/*-------------------------------------------------------------*/
#include <stdioh>
#include <stdlibh>
#include <stringh>
#include <sqlh>
#include <talh>
#include <cextdecs>
#pragma list
#define MAXSUBSORTS  3      /* max number of subsorts          */
#define MAXSCRATCHVOLS 4    /* max number of scratch volumes   */
char  home_term_name[48];   /* terminal name                   */
short home_term_filenum;    /* file number                     */
short home_term_len;        /* actual len of hometerm name     */
short home_term_maxlen = 48;/* max len of hometerm name        */
short error_detail;         /* output from process_getinfo_    */
/*-------------------------------------------------------------*/
/* FastSort control and flags information.                     */
/*-------------------------------------------------------------*/
_lowmem short ctlblk[200];  /* control block for sort interface*/
_lowmem short key_array[4]; /* SORTMERGESTART key field defns  */
short sflag1 = 1;     /* use 22-word SORTMERGESTATISTICS array */
short flags = 512;    /* use expanded process_start structure  */
                      /* same as setting flags.<6> in TAL      */
/*-------------------------------------------------------------*/
/* FastSort error and statistics variables.                    */
/*-------------------------------------------------------------*/
short  error;                      /* error return parameter   */
_lowmem short error_buf[20],       /* error message buffer     */
              error_source[20],    /* error related info       */
              sub_index,           /* subsort that caused error*/
              sub_cpu_pin;         /* CPU,PIN of this subsort  */
_lowmem long  error_code[40]; /* Fastsort & system error codes */
struct sortstats_template {
   short maxrecordsize;
   short bufferpages;
   long  records;
   long  elapsedtime;
   long  compares;
   long  scratchseeks;
   long  iowaittime;
   long  scratchfileeof;
   long  initialruns;
   short firstmergeorder;
   short mergeorder;
   short intermediatepasses;
   long  numberofduplicates;
   } _lowmem sortstats;
```

Example 6-1. C Example of a Parallel Sort Run (page 2 of 5)

```
struct newprocess_parms_template {  /* 29-word structure       */
   short priority;
   short memory;
   short cpu;
   short system;
   short segment_size;
   char  swap_file[24];
   char  program_file[24];
   } _lowmem newprocess_parms[MAXSUBSORTS + 1];
/*-------------------------------------------------------------*/
/* Input and output files                                      */
/*-------------------------------------------------------------*/
_lowmem char  infile[]  = "$DATA2 FSORT256INFILE";
_lowmem char  outfile[] = "$DATA2 FSORT256OUTFIL";

struct scratch_files_template {
  char filename[24];
} _lowmem scratch_files[MAXSUBSORTS + 1];

  struct scratch_pool_template {
  short reserved_word1;
  short reserved_word2;
  short reserved_word3;
  short reserved_word4;
  short reserved_word5;
  short reserved_word6;
  short reserved_word7;
  short reserved_word8;short use_scratch;
  short num_scratch_vols;
  short scratchvolnames[MAXSCRATCHVOLS];
} _lowmem scratch_pool;

void error_handler (void);
short DisplaySortStatistics (struct sortstats_template *);
/*-------------------------------------------------------------*/
#pragma page  " Main logic "
/*-------------------------------------------------------------*/

int main (void)
{
    short errlen = 0;
    short i,j;
/* Initialize temp scratch volume name and I/O filename arrays.*/
/* Leave blank for distributor-collector sort.                 */
    char tmp_dist_scr = "                           ";
    char tmp_scr1[] = "$DATA1                     ";
    char tmp_scr2[] = "$DATA2                     ";
    char tmp_scr3[] = "$DATA4                     ";
    char tmp_swap[] = "                           ";
    char tmp_prog[] = "                           ";
    char tmp_infile[] =  "$DATA2 FSORT256INFILE   ";
    char tmp_outfile[] = "$DATA2 FSORT256OUTFIL   ";
/*initialize scratch pool array;will be passed to all subsorts */
    char tmp_scr_pool[] = "$DATA1  $DATA2  $DATA3  $DATA4  ";
    _lowmem short actuallen;    /* size of statistics in words */
/*-------------------------------------------------------------*/
/* Perform standard initialization.                            */
/*-------------------------------------------------------------*/

   error = PROCESS_GETINFO_(,,,,,,(char *)&home_term_name,
                           home_term_maxlen,
                           &home_term_len,
                           ,,,,,,,,,,,&error_detail);
   if (error)
     DEBUG;
  if (FILE_OPEN_(home_term_name,
                 home_term_len,
                 &home_term_filenum) != CCE )
     DEBUG;
   INITIALIZER;                    /* read the startup message */
```

Example 6-1. C Example of a Parallel Sort Run (page 3 of 5)

```
/*-------------------------------------------------------------*/
/* Initialize SORT key definitions array.                      */
/*-------------------------------------------------------------*/
  key_array[0] = 1;  /* number of keys                         */
  key_array[1] = 9;  /* definition = binary,unsigned,ascending */
  key_array[2] = 2;  /* key length = 2 bytes                   */
  key_array[3] = 0;  /* key offset = 0 bytes                   */
/*-------------------------------------------------------------*/
/* Initialize structures to start SORTPROG with parallel option*/
/*-------------------------------------------------------------*/
  for (i = 0; i <= MAXSUBSORTS; i++)
    {
     newprocess_parms[i].priority     = 0;
     newprocess_parms[i].memory       = 1;
     newprocess_parms[i].system       = 2;
     newprocess_parms[i].segment_size = 3;
     memcpy(&newprocess_parms[i].swap_file[0],&tmp_swap[0],24);
     memcpy(&newprocess_parms[i].program_file[0],&tmp_prog[0],24);
    }
/*-------------------------------------------------------------*/
/* Set CPU numbers and scratch file names                      */
/*-------------------------------------------------------------*/
     newprocess_parms[0].cpu          = 0;
     newprocess_parms[1].cpu          = 1;
     newprocess_parms[2].cpu          = 2;
     newprocess_parms[3].cpu          = 3;
     memcpy(&scratch_files[0].filename[0],&tmp_dist_scr[0],24);
     memcpy(&scratch_files[1].filename[0],&tmp_scr1[0],24);
     memcpy(&scratch_files[2].filename[0],&tmp_scr2[0],24);
     memcpy(&scratch_files[3].filename[0],&tmp_scr3[0],24);
     memcpy(&infile[0],&tmp_infile[0],24);
     memcpy(&outfile[0],&tmp_outfile[0],24);
/*-------------------------------------------------------------*/
/* Set SCRATCH and SCRATCHON volume names. Tell SORTPROG to    */
/* use specified volumes for scratch.                          */
/*-------------------------------------------------------------*/
     scratchpool.use_scratch = 0;
     scratch_pool.num_scratch_vols = 4;
     memcpy(&scratch_pool.scratch_vol_names[0],
            &tmp_scr_pool[0],32);
/*-------------------------------------------------------------*/
/* Call SORTBUILDPARM to initialialize SORTPROG control block. */
/*-------------------------------------------------------------*/
  error = SORTBUILDPARM (&ctlblk[0]
                        ,
                        ,
                        ,
                        ,
                        ,
                        ,
                        ,
                        ,
                        ,
                        ,
                        ,(short *) &scratch_pool
                        );
  if (error)                 /* check for SORTBUILDPARM error  */
     {
       errlen = SORTERRORSUM (&ctlblk[0],
                               &error_buf[0],
                               &error_code[0],
                               &error_source[0]);
     error_handler;
     return EXIT_FAILURE;
     }
```

Example 6-1. C Example of a Parallel Sort Run (page 4 of 5)

```
/*--------------------------------------------------------------*/
/* Call SORTMERGESTART to start the SORTPROG processes.         */
/*--------------------------------------------------------------*/
  error = SORTMERGESTART
                  (&ctlblk[0],
                   &key_array[0],,1,
                   (short *) &infile[0],,,,,
                   (short *) &outfile[0],,,
                   flags,,,
                   (short *) &scratch_files[0].filename[0],,
                   (short *) &newprocess_parms[0].priority,,,,,,
                   MAXSUBSORTS);
  if (error)                /* check for SORTMERGESTART error */
     {
       errlen = SORTERRORSUM (&ctlblk[0],
                             &error_buf[0],
                             &error_code[0],
                             &error_source[0],
                             &sub_index,&sub_cpu_pin);
     error_handler;
     return EXIT_FAILURE;
     }
/*--------------------------------------------------------------*/
/* Return SORTPROG completion errlen and statistics. Set        */
/* length in words, to return all statistics information.       */
/*--------------------------------------------------------------*/
  actuallen = sizeof(sortstats)/2;
  error = SORTMERGESTATISTICS (&ctlblk[0], &actuallen,
                              (short *) &sortstats,sflag1);
  if (error)          /* check for SORTMERGESTATISTICS error  */
       {
       errlen = SORTERRORSUM (&ctlblk[0],
                             &error_buf[0],
                             &error_code[0],
                             &error_source[0]);
       error_handler;
       return EXIT_FAILURE;
       }
/*--------------------------------------------------------------*/
/* Call function to display the statistics                      */
/*--------------------------------------------------------------*/
  error = DisplaySortStatistics (&sortstats);
  if (error)
       return EXIT_FAILURE;
/*--------------------------------------------------------------*/
/* CALL SORTMERGEFINISH to stop SORTPROG after the process      */
/* successfully completes the current sort and merge run(s).    */
/*--------------------------------------------------------------*/
    error = SORTMERGEFINISH (&ctlblk[0]);
    if (error)          /* check for SORTMERGEFINISH error    */
       {
       errlen = SORTERRORSUM (&ctlblk[0],
                             &error_buf[0],
                             &error_code[0],
                             &error_source[0]);
       error_handler;
       return EXIT_FAILURE;
       }
    FILE_CLOSE_ (home_term_filenum);
}                                    /* End of Main logic      */
```

Example 6-1. C Example of a Parallel Sort Run (page 5 of 5)

```
void error_handler (void)
{
 /* error handling stubbed out */
  return;
}

short DisplaySortStatistics (struct sortstats_template *instats)
{
 /* Printing of statistics stubbed out */
  return EXIT_SUCCESS;
}
/*-----------------------E-N-D--------------------------------*/
```

Parallel Sorting From COBOL85 Programs

Example 6-2 on page 6-20 shows a COBOL85 program that calls COBOL85 interface routines to perform a parallel sort run.

Example 6-2. COBOL85 Example of a Parallel Sort Run (page 1 of 3)

```
*------------------------------------------------------------
*           FastSort Parallel Sort Run
* This program calls the COBOL85 interface routines
* COBOL85^SET^SORT^PARAM^VALUE and
* COBOL85^SET^SORT^PARAM^TEXT to start a parallel sort run.
*------------------------------------------------------------
?SYMBOLS, INSPECT
?LIBRARY $SYSTEM.SYSTEM.COBOLLIB
?LIBRARY $SYSTEM.SYSTEM.CBL85UTL
 IDENTIFICATION DIVISION.
 PROGRAM-ID.    PARALLEL-SORT-EXAMPLE.
 ENVIRONMENT DIVISION.
   CONFIGURATION SECTION.
   INPUT-OUTPUT SECTION.
   FILE-CONTROL.
   SELECT OUTPUT-FILE
          ASSIGN TO "=OUTFILE"
          ORGANIZATION IS SEQUENTIAL
          ACCESS MODE IS SEQUENTIAL.
   SELECT SORT-FILE
     ASSIGN TO "SORTFILE".
 DATA DIVISION.
 FILE SECTION.
 FD OUTPUT-FILE
     LABEL RECORDS ARE OMITTED
     RECORD CONTAINS 19 CHARACTERS.
 01  OUT-RECORD.
     05 SORT-RECORD-NO    PIC 9(4).
     05 FILLER            PIC X(5).
     05 SORT-CODE         PIC X(10).
SD  SORT-FILE
     RECORD CONTAINS 19 CHARACTERS.
 01  SORT-RECORD.
     05 SORT-RECORD-NO    PIC 9(4).
     05 FILLER            PIC X(5).
     05 SORT-CODE         PIC X(10).
WORKING-STORAGE SECTION.
 01 FLAGS.
    05 MORE-OUTPUT-FLAG       PIC X(3) VALUE "YES".
       88 MORE-OUTPUT                  VALUE "YES".
       88 NO-MORE-OUTPUT               VALUE "NO".
 01  RETURN-CODE              PIC 99 COMP.
 01  OUTPUT-COUNTER           PIC 9999 COMP VALUE 0.
 01  INPUT-RECORDS            PIC 9999      VALUE 1000.
 01  NUMBER-OF-SUBSORTS       PIC 99        VALUE 2.
 01  VALUE-PARAM              PIC X(20).
 01  SCRATCH-FILE-1           PIC X(8) VALUE "SCRATCH1".
 01  SCRATCH-FILE-2           PIC X(8) VALUE "SCRATCH2".
 01  WS-ORDR-CODE.
     05 WS-RECORD-NO          PIC 9(4) VALUE 0.
     05 FILLER                PIC X(5) VALUE SPACES.
     05 WS-CODE.
        06  WS-CODE-NBR       PIC 9999 VALUE 1000.
        06  WS-CODE-FIL       PIC X(6).
*------------------------------------------------------------
 PROCEDURE DIVISION.
 MAIN SECTION.
     OPEN OUTPUT OUTPUT-FILE.
     DISPLAY "Starting FastSort parallel sort run..."
     PERFORM SORT-RECORDS.
     DISPLAY "FastSort parallel sort run completed."
     STOP RUN.
 SORT-RECORDS SECTION.
```

Example 6-2. COBOL85 Example of a Parallel Sort Run (page 2 of 3)

```
*------------------------------------------------------------
* Specify the number of subsort processes.
*------------------------------------------------------------
     MOVE "SUBSORT-COUNT" TO VALUE-PARAM.
     ENTER "COBOL85^SET^SORT^PARAM^VALUE"
           USING  SORT-FILE,
                  VALUE-PARAM, NUMBER-OF-SUBSORTS
           GIVING RETURN-CODE.
     IF RETURN-CODE NOT = 0
         PERFORM ERROR-RETURN
     END-IF.
*------------------------------------------------------------
* Specify the number of input records to sort.
*------------------------------------------------------------
 MOVE "IN-FILE-COUNT" TO VALUE-PARAM.
 ENTER "COBOL85^SET^SORT^PARAM^VALUE"
       USING  SORT-FILE,
              VALUE-PARAM, INPUT-RECORDS
       GIVING RETURN-CODE.
 IF RETURN-CODE NOT = 0
     PERFORM ERROR-RETURN
 END-IF.
*------------------------------------------------------------
* Specify a scratch file for each subsort process.
*------------------------------------------------------------
 MOVE "SCRATCH-FILE" TO VALUE-PARAM.
 ENTER "COBOL85^SET^SORT^PARAM^TEXT"
       USING  SORT-FILE,
              VALUE-PARAM, SCRATCH-FILE-1, 1
       GIVING RETURN-CODE.
 IF RETURN-CODE NOT = 0
     PERFORM ERROR-RETURN
 END-IF.

 ENTER "COBOL85^SET^SORT^PARAM^TEXT"
           USING  SORT-FILE,
                  VALUE-PARAM, SCRATCH-FILE-2, 2
           GIVING RETURN-CODE.
     IF RETURN-CODE NOT = 0
         PERFORM ERROR-RETURN
     END-IF.
*------------------------------------------------------------
*  Execute the sort run.
*------------------------------------------------------------
 SORT SORT-FILE ON ASCENDING KEY SORT-CODE OF SORT-RECORD
      INPUT PROCEDURE  INPUT-SECTION
      OUTPUT PROCEDURE OUTPUT-SECTION.
*------------------------------------------------------------
* Display any FastSort error codes.
*------------------------------------------------------------
 ERROR-RETURN SECTION.
 ERR-RTN.
     DISPLAY "Error-Return Code is: " RETURN-CODE.
     STOP RUN.
*------------------------------------------------------------
* Generate the input file.  (Note:  An actual program would
* get input records from an existing file or a process.)
*------------------------------------------------------------
 INPUT-SECTION SECTION.
 INPUT-ROUTINE.
     DISPLAY "Input sort procedure entered...".
     PERFORM INPUT-RECORDS TIMES
        ADD 1 TO WS-RECORD-NO
        SUBTRACT 1 FROM WS-CODE-NBR
        RELEASE SORT-RECORD FROM WS-ORDR-CODE
     END-PERFORM.
     DISPLAY "Input sort records created: " INPUT-RECORDS.
```

Example 6-2. COBOL85 Example of a Parallel Sort Run (page 3 of 3)

```
*------------------------------------------------------------
* Return the sorted records.
*------------------------------------------------------------
 OUTPUT-SECTION SECTION.
 OUTPUT-ROUTINE.
     DISPLAY "Output sort procedure entered...".
     SET MORE-OUTPUT TO TRUE.
     PERFORM UNTIL NO-MORE-OUTPUT
         RETURN SORT-FILE
           AT END SET NO-MORE-OUTPUT TO TRUE
             NOT AT END
               MOVE CORRESPONDING SORT-RECORD TO OUT-RECORD
               WRITE OUT-RECORD
               ADD 1 TO OUTPUT-COUNTER
         END-RETURN
     END-PERFORM.
     DISPLAY "Output sort records returned: " OUTPUT-COUNTER.
```

Parallel Sorting From TAL Programs

Example 6-3 on page 6-23 shows a TAL example that sets up a parallel sort run with three subsort processes.

Example 6-3. TAL Example of a Parallel Sort Run (page 1 of 3)

```
?SYMBOLS, NOCODE, INSPECT, MAP, LMAP, DATAPAGES 64
!------------------------------------------------------------!
!                 FastSort Parallel Sort Run                 !
! This program uses 3 subsorts to sort an input file.        !
!------------------------------------------------------------!
! Global Declarations.                                       !
!------------------------------------------------------------!
INT .home^term^name[0:11] := 12*["  "];  ! Name
INT  home^term^filenum;                  ! File number
INT .blank^name[0:11] := 12*["  "];  ! Blank file name
!------------------------------------------------------------!
! Subsort information.                                       !
!------------------------------------------------------------!
LITERAL max^subsort = 3;  ! Maximum number of subsorts
INT .ctlblk[0:199];       ! Control block for sort interface
INT .key^array[0:3]; ! Sort key array definitions
!------------------------------------------------------------!
! Input and output files.                                    !
!------------------------------------------------------------!
INT .in^file[0:11]  := ["$DISK01 FASTSORTINFILE  "];
INT .out^file[0:11] := ["$DISK01 FASTSORTOUTFILE "];
!------------------------------------------------------------!
! SORTPROG new process information.                          !
!------------------------------------------------------------!
STRUCT .newprocess^parms[0:max^subsort];
   BEGIN
   INT priority;           ! Priority
   INT memory;             ! Memory (ignored by FastSort)
   INT cpu;                ! CPU number
   INT system;             ! System
   INT segment^size;       ! Extended memory
   INT swap^file[0:11];    ! Swap file name
   INT program^file[0:11]; ! Program file name
   END;
INT flags := %B0000001000000000; ! Sort for flags newprocess
!------------------------------------------------------------!
! Scratch files.                                             !
!------------------------------------------------------------!
STRUCT .scratch^files[0:max^subsort];
   BEGIN
   INT filename[0:11];   ! Scratch file name
   END;
!------------------------------------------------------------!
! Sort error and statistics variables.                       !
!------------------------------------------------------------!
INT     .error^buf[0:31],     ! Error message
         error^source,        ! Error-related information
         sub^index,           ! Subsort that caused an error
         sub^cpu^pin,         ! CPU,PIN of this subsort
         error,               ! Return error code
        .stat[0:20];          ! Buffer for statistics
INT(32)  error^code;          ! FastSort and system error codes
!------------------------------------------------------------!
?PAGE "External Declarations from EXTDECS0"
!------------------------------------------------------------!
?NOLIST
?SOURCE $SYSTEM.SYSTEM.EXTDECS0 (DEBUG,
?                                 INITIALIZER,
?                                 MYTERM,
?                                 OPEN,
?                                 READ,
?                                 SORTMERGESTART,
?                                 SORTERRORSUM,
?                                 SORTMERGESTATISTICS,
?                                 SORTMERGEFINISH,
?                                 STOP)
```

Example 6-3. TAL Example of a Parallel Sort Run (page 2 of 3)

```
!-----------------------------------------------------------!
?PAGE "MAIN Procedure"
!-----------------------------------------------------------!
PROC main^proc MAIN;
BEGIN
  INT length;
!-----------------------------------------------------------!
! Open the home terminal and call the                       !
! INITIALIZER to read the startup message.                  !
!-----------------------------------------------------------!
CALL MYTERM (home^term^name);
CALL OPEN (home^term^name, home^term^filenum);
IF <> THEN CALL DEBUG;
CALL INITIALIZER;
!-----------------------------------------------------------!
! Initialize the sort key array definitions.                !
!-----------------------------------------------------------!
key^array[0] := 1; ! Number of keys
key^array[1] := 9; ! Key descriptor: binary signed, ascending
key^array[2] := 2; ! Key length: 2 bytes
key^array[3] := 0; ! Key offset: 0 bytes
!-----------------------------------------------------------!
! Start SORTPROG process with the parallel option.          !
!-----------------------------------------------------------!
USE i;
FOR i := 0 TO max^subsort DO
  BEGIN     ! Fill default values
  newprocess^parms[i].priority     := -1;
  newprocess^parms[i].memory       := -1;
  newprocess^parms[i].system       := -1;
  newprocess^parms[i].segment^size := -1;
  newprocess^parms[i].swap^file    ':=' blank^name FOR 12;
  newprocess^parms[i].program^file ':=' blank^name FOR 12;
  END;
!-----------------------------------------------------------!
! Set CPU numbers and scratch file names.                   !
!-----------------------------------------------------------!
newprocess^parms[0].cpu := 1;
newprocess^parms[1].cpu := 3;
newprocess^parms[2].cpu := 4;
newprocess^parms[3].cpu := 6;
scratch^files[0].filename ':=' "                 ";
scratch^files[1].filename ':=' "$DISK01          ";
scratch^files[2].filename ':=' "$DISK01          ";
scratch^files[3].filename ':=' "$DISK01          ";
!-----------------------------------------------------------!
! Call SORTMERGESTART procedure.                            !
!-----------------------------------------------------------!
error := SORTMERGESTART (ctlblk,
                         key^array,,1,
                         in^file,,,,,
                         out^file,,,
                         flags,,,
                         scratch^files[0].filename,,
                         newprocess^parms[0].priority,,,,,,
                         max^subsort);

IF error THEN ! Check for SORTMERGESTART error.
   BEGIN
   length := SORTERRORSUM (ctlblk,
                           error^buf, error^code,
                           error^source,
                           sub^index, sub^cpu^pin);
   ! Process the SORTMERGESTART error.
   END;
```

Example 6-3. TAL Example of a Parallel Sort Run (page 3 of 3)

```
!------------------------------------------------------------!
! Return statistics and check the sort completion.           !
!------------------------------------------------------------!
error := SORTMERGESTATISTICS (ctlblk,length,stat);
IF error THEN ! Check for SORTMERGESTATISTICS error.
   BEGIN
   length := SORTERRORSUM (ctlblk,
                           error^buf, error^code,
                           error^source,
                           sub^index, sub^cpu^pin);
   ! Process the SORTMERGESTATISTICS error.
   END;
END;                 ! of the MAIN procedure !
!------------------------------------------------------------!
```

7 Using SORT and SUBSORT DEFINEs

Before you start a sort or merge run, you can set or change operating system parameters that affect FastSort. While the FastSort default settings are often sufficient for small sort or merge runs, modifying the default settings can improve the performance of a large run.

You modify default settings for a sort operation with CLASS SORT and SUBSORT DEFINEs. You can set or change a DEFINE either interactively or programmatically. Tasks you can perform with SORT and SUBSORT DEFINEs include:

- Select a less busy processor for SORTPROG processes
- Select a disk with more free space than the default for your initial scratch file
- Manually specify certain parameters that are difficult to set from an application program
- Specify parameters for sorting from SQL/MP and other products that do not allow you to specify FastSort parameters at run time

This section describes the following FastSort DEFINEs:

`SORT DEFINE`

is the FastSort DEFINE you use to control the SORTPROG process in a serial sort run or the distributor-collector SORTPROG process in a parallel sort run.

`SUBSORT DEFINE`

is the FastSort DEFINE you use to control a subsort process in a parallel sort run. You can specify 2 to 8 SUBSORT DEFINEs to set parameters for each subsort process that is linked to a specific SORT DEFINE.

`=_SORT_DEFAULTS DEFINE`

is the FastSort default DEFINE. You use this DEFINE to specify FastSort parameters for SQL/MP and applications that otherwise cannot set the parameters. Section 8, Sorting From NonStop SQL/MP contains additional information about configuring sorts from SQL/MP.

Determining the Precedence of DEFINEs

There are four possible ways to specify FastSort parameter values. If a conflict occurs between two or more values, FastSort chooses a value as follows:

Priority	**Value Type**	**Specified By**
1	User-specified	A SORT or SUBSORT DEFINE
2	User parameter	A FastSort interactive parameter or system procedure call
3	Default	The =_SORT_DEFAULTS DEFINE

Use a single method to specify values for a subsort process. For subsort processes, FastSort treats all parameter values as a single entity. After FastSort determines the source of information for a subsort process, it ignores values from other sources.

Setting DEFINE Attributes

The table below lists the attributes for CLASS SORT and SUBSORT DEFINEs. These attributes are described in this section and in Section 3, Using FastSort Commands.

Class	**Attributes**
SORT	BLOCK, CPU, CPUS, MODE, NOSCRATCHON, NOTCPUS, PRI, PROGRAM, SCRATCH, SCRATCHON, SEGMENT, SUBSORTS, SWAP, VLM
SUBSORT	BLOCK, CPU, PRI, PROGRAM, SCRATCH, SEGMENT, SWAP

Setting SORT DEFINE Attributes

The following SORT DEFINE attributes correspond to parameters for the FastSort interactive commands in Section 3, Using FastSort Commands. Only the SCRATCH attribute is required; all other SORT DEFINE attributes are optional. To set parameters for SQL/MP, see Creating and Using the =_SORT_DEFAULTS DEFINE on page 7-13.

`BLOCK` *`size`*

specifies the block size, in bytes, for scratch files. Specify any multiple of 2048 bytes up to 56 KB, such as:

```
SET DEFINE BLOCK 28672
```

The number you specify must at least equal the size of the largest input record, rounded up to the nearest even byte, plus 14 bytes overhead. For optimal performance, specify 56 KB for local scratch files and 28 KB for remote scratch files. The default is 56 KB.

`CPU` *`processor`*

specifies the processor (CPU) number for the SORTPROG process. The range is 0 through 15, such as:

```
SET DEFINE CPU 2
```

The default is the CPU where the process that starts SORTPROG is running.

`CPUS {` *`processor`* `[,` *`processor`* `]... | ALL }`

specifies the processor (CPU) numbers that are available for subsort processes, The range is 0 through 15. A value of ALL means that all processors are available for subsorts. To specify a list of processors, enclose the numbers in parentheses and separate them with commas, as follows:

```
SET DEFINE CPUS ALL
SET DEFINE CPUS 5
SET DEFINE CPUS (1,3,4)
```

You can specify the CPUS and NOTCPUS attributes in the same DEFINE.

`MODE { MINTIME | MINSPACE | AUTOMATIC }`

For a description of MINSPACE, MINTIME, and AUTOMATIC, see RUN Command on page 3-19. An example of MODE attribute syntax is:

```
SET DEFINE MODE MINSPACE
```

`NOSCRATCHON (`*`volume-name`* `[,` *`volume-name`* `]...)`

specifies volumes that FastSort should not use for overflow scratch files. If the scratch file specified in the SCRATCH attribute becomes full and no SCRATCHON values exist, FastSort tries to create an overflow scratch file on a volume not specified in the NOSCRATCHON attribute, protected by the Safeguard product, $SYSTEM, or a TMF audit trail disk. Enclose NOSCRATCHON volume names in parentheses and separate with commas, as follows:

```
SET DEFINE NOSCRATCHON ( $data2 , $data3 )
```

FastSort recognizes the wild-card characters * and ? for the NOSCRATCHON attribute. See the description of SCRATCHON following for examples of how to use these characters.

You can specify up to 32 NOSCRATCHON volumes. If you specify NOSCRATCHON, you cannot specify SCRATCHON. Note that this attribute requires up to 276 additional bytes of stack space.

`NOTCPUS {` *`processor`* `[,` *`processor`* `]... }`

specifies the processor (CPU) numbers that are not available for subsort processes. The range is 0 through 15. To specify a list of processors, enclose the numbers in parentheses and separate them with commas, as follows:

```
SET DEFINE NOTCPUS (2,5,6)
```

`PRI` *`priority`*

specifies the priority to assign to the SORTPROG process. The range is 1 through 199, such as:

```
SET DEFINE PRI 120
```

The default is the operating system priority for the parent process.

`PROGRAM` *`file-name`*

specifies a local or remote program file name to run in place of the default program file, such as:

```
SET DEFINE PROGRAM $data.fastsort.sortprog
```

`SCRATCH` *`file-name`*

specifies a disk file name or disk volume name for an initial scratch file. This attribute is required. For example:

```
SET DEFINE SCRATCH $data.fastsort.scratch
```

If the file already exists, it must be unstructured. If the initial scratch file becomes full, then either the SCRATCHON or NOSCRATCHON attribute determines the volume for an overflow scratch file.

`SCRATCHON (`*`volume-name`*` [, `*`volume-name`*` ]...)`

specifies volumes that FastSort should use for overflow scratch files. If the volume specified in the SCRATCH attribute becomes full, FastSort tries to create additional overflow scratch files on a SCRATCHON volume. To specify a list of volume names, enclose the names in parentheses and separate with commas, as follows:

```
SET DEFINE SCRATCHON ( $data4 , $data5 )
```

FastSort recognizes the wild-card characters * and ? for the SCRATCHON attribute. For example:

```
SET DEFINE SCRATCHON $data*
```

specifies as available for overflow scratch files all volumes whose names begin with the string *`$data`*.

```
SET DEFINE SCRATCHON $data?
```

specifies as available for overflow scratch files those volumes whose names begin with the string *`$data`* and contain a single trailing character.

You can specify up to 31 SCRATCHON volumes. If you specify SCRATCHON, you cannot specify NOSCRATCHON. Note that this attribute requires up to 276 additional bytes of stack space.

`SEGMENT` *`size`*

specifies the size in pages of the extended data segment for FastSort, such as:

```
SET DEFINE SEGMENT 256
```

The value must be at least 256 pages but must not represent more than 90 percent of available memory. If you specify SEGMENT, you must omit MODE. The default is the same as MODE AUTOMATIC.

`SUBSORTS ( `*`DEFINE-name`*` [, `*`DEFINE-name`*` ]... )`

specifies a list of DEFINE names for subsort processes. Separate the DEFINE names with commas and enclose them in parentheses, such as:

```
SET DEFINE SUBSORTS (=subsorta, =subsortb, =subsortc)
```

FastSort checks DEFINE names for validity at run time.

`SWAP `*`file-name`*

specifies the name of a swap file to use in an extended memory data segment. The swap file you specify must be a disk file or volume on the local node, such as:

```
SET DEFINE SWAP $data.fastsort.swapfile
```

If the file already exists, it must be unstructured. The default location for the swap file depends on the location of the scratch file. If the scratch file is local, the swap file is on the scratch volume. For remote scratch files, the default is the volume where the program file is running.

`VLM { ON | OFF }`

specifies whether to use additional extended memory during a sort run. For example:

```
SET DEFINE VLM ON
```

makes extra memory available. FastSort uses the additional extended memory to either complete the sort in a single pass or store partial information until the sort is complete. When VLM is ON, FastSort uses up to 127.5 MB of extended memory, if available. The default value is OFF.

Setting SUBSORT DEFINE Attributes

The SUBSORT DEFINE attributes set values for a subsort process that you name in the SUBSORT attribute of a SORT DEFINE. SUBSORT DEFINE attributes correspond to the parameters of the SUBSORT command, which are described in Section 3, Using FastSort Commands. To set parameters for parallel SQL/MP load operations, see Loading Data on page 8-7.

`CPU `*`processor`*

specifies the number of the processor (CPU) in which to run the subsort process, such as:

```
SET DEFINE CPU 3
```

The range is 0 through 15. The default is the processor in which the primary disk process for the initial scratch volume is running.

`PRI` *`priority`*

specifies the priority for the subsort process. The range is 1 through 199. The default is the operating system priority for the parent process.

```
SET DEFINE PRI 180
```

`PROGRAM` *`file-name`*

specifies a local or remote program file name to run for the subsort process in place of the default program file, such as:

```
SET DEFINE PROGRAM $data.another.sortprog
```

`SCRATCH` *`file-name`*

specifies a disk file name or disk volume name for an initial scratch file for the subsort process. Specify a unique scratch file for each subsort process. For example:

```
SET DEFINE SCRATCH $data.temp1
```

If the file already exists, it must be unstructured. If the subsort scratch volume becomes full, then either NOSCRATCHON or the SCRATCHON attribute of the distributor-collector process determines the volume for another subsort scratch file. If no values are specified for the SCRATCHON and NOSCRATCHON attributes, then FastSort uses volume characteristics to select an overflow scratch volume. For more information about scratch, see Managing Sort Workspace on page 9-1.

`SEGMENT` *`size`*

specifies the extended data segment size in pages for the subsort process. For example:

```
SET DEFINE SEGMENT 256
```

The value must be at least 256 pages but must not represent more than 90 percent of available memory. The default value is 256 pages.

`SWAP` *`file-name`*

is the name of a swap file for the subsort process. The value you specify must be a local disk file or disk volume, such as:

```
SET DEFINE SWAP $data.temp4
```

If the file already exists, it must be unstructured. The default location for the swap file depends on the location of the scratch file. If the scratch file is local, the swap file is on the scratch volume. For remote scratch files, the default is the volume where the program file is running.

Creating and Using DEFINEs Interactively

Use the TACL DEFINE commands listed below to interactively create and modify SORT and SUBSORT DEFINEs. The operating system places the DEFINEs in the process file segment (PFS) of your TACL process.

Command	**Description**
ADD DEFINE	Creates a DEFINE in the PFS of the current TACL process.
ALTER DEFINE	Changes the attribute settings of an existing DEFINE in the PFS.
DELETE DEFINE	Deletes one or more existing DEFINEs from the PFS.
INFO DEFINE	Displays the attributes and their settings for one or more existing DEFINEs.
RESET DEFINE	Resets the attributes of one or more existing DEFINE to their initial values.
SET DEFINE	Sets the values of one or more existing DEFINE attributes in the working set.
SHOW DEFINE	Displays a value of a specific DEFINE attribute, the values of all attributes, or the values of all attributes in the working set.

For additional information, including the syntax, for these commands, see the *TACL Reference Manual*.

Enabling DEFINEs

You must set the TACL DEFMODE variable to ON before you can use a DEFINE. If the DEFMODE variable is OFF, DEFINEs do not affect FastSort or any other processes. To determine the current value of the DEFMODE variable, use the SHOW DEFMODE command. To enable DEFINEs, use the SET DEFMODE ON command.

Creating a SORT DEFINE

The following example creates a SORT DEFINE named =DISTRIBUTOR_COLLECTOR and three SUBSORT DEFINEs named =SUBSORTA, =SUBSORTB, and =SUBSORTC. The SET DEFINE CLASS SORT establishes the initial working attribute set and their default values. TACL automatically assigns these default values to the working attribute set in the next SORT DEFINE you create. You can issue additional SET DEFINE commands to set other attributes in the working attribute set. Then use ADD DEFINE to create and name the =DISTRIBUTOR_COLLECTOR based on the working attribute set, as follows:

```
SET DEFINE CLASS SORT
SET DEFINE SWAP  $disk.fastsort.swapfile
SET DEFINE MODE MINSPACE
SET DEFINE SUBSORTS (=subsorta,=subsortb,=subsortc)
ADD DEFINE =distributor_collector
```

You can also create the same SORT DEFINE using a single command as shown in the next example. The ampersand (&) is the continuation character for a TACL command that continues on the next physical line.

```
ADD DEFINE =distributor_collector, CLASS SORT,   &
    SCRATCH $disk.fastsort.scratch,              &
    SWAP $disk.fastsort.swapfile,                &
    MODE MINSPACE,                               &
    SUBSORTS (=subsorta,=subsortb,=subsortc)
```

Displaying a DEFINE

Use the INFO DEFINE command with the DETAIL option to display the attributes and values of one or more DEFINEs. Use the SHOW DEFINE command to display the values of specific attributes. For example, to display the attributes and values for the =DISTRIBUTOR_COLLECTOR DEFINE, enter:

```
INFO DEFINE =distributor_collector, DETAIL
```

TACL displays:

```
    Define Name           =DISTRIBUTOR_COLLECTOR
    CLASS                 SORT
    SWAP                  $DISK.FASTSORT.SWAPFILE
    MODE                  MINSPACE
    SUBSORTS              (=SUBSORTA,=SUBSORTB,=SUBSORTC)
```

The SHOW DEFINE command displays current attributes and attribute values. For example, to display the working attribute set with current values, enter:

```
SHOW DEFINE *
```

TACL displays:

```
    CLASS                 SORT
    SCRATCH
    SWAP                  $DISK.FASTSORT.SWAPFILE
    MODE                  MINSPACE
    CPU
    BLOCK
    PRI
    SEGMENT
    PROGRAM
    CPUS
    NOTCPUS
    SUBSORTS              (=SUBSORTA,=SUBSORTB,=SUBSORTC)
    VLM
    NOSCRATCHON
    SCRATCHON
```

For more information about the SHOW DEFINE command, see the example under Examples of SORT and SUBSORT DEFINEs on page 7-15, and the *TACL Reference Manual*.

Creating a SUBSORT DEFINE

A SUBSORT DEFINE controls a subsort process in a parallel sort run. Specify between 2 and 8 SUBSORT DEFINEs for a SORT DEFINE.

Note. FastSort supports up to 16 SUBSORT DEFINEs; however, to prevent run-time errors and performance problems, HP recommends that you specify no more than 8 SUBSORT DEFINEs.

You create a SUBSORT DEFINE in the same manner as a SORT DEFINE. To use a SUBSORT DEFINE, you must also name the SUBSORT DEFINE in the SORT DEFINE SUBSORTS attribute. At run time, each SUBSORTS attribute of a SORT DEFINE must correspond to an existing SUBSORT DEFINE.

The following example creates a SUBSORT DEFINE named =SUBSORTA, which is associated with the =DISTRIBUTOR_COLLECTOR SORT DEFINE:

```
SET DEFINE CLASS SUBSORT
SET DEFINE SCRATCH  $disk.temp1
SET DEFINE SWAP     $disk.temp2
ADD DEFINE =subsorta
```

An INFO DEFINE command for =SUBSORTA displays:

```
    Define Name        =SUBSORTA
    CLASS              SUBSORT
    SCRATCH            $DISK.TEMP1
    SWAP               $DISK.TEMP2
```

Modifying a DEFINE

You can also use TACL commands to add, modify, or delete the attributes of a SORT or SUBSORT DEFINE (or the entire DEFINE). The following examples show several ALTER DEFINE and RESET DEFINE commands with the results displayed with the INFO DEFINE command.

To add one or more SUBSORT DEFINE names to an existing SORT DEFINE, use the ALTER DEFINE command:

```
ALTER DEFINE =distributor_collector,          &
      SUBSORTS (=subsorta,=subsortb,=subsortc,=subsortd)

INFO DEFINE =distributor_collector, DETAIL
    Define Name     =DISTRIBUTOR_COLLECTOR DEFINE
    CLASS           SORT
    SWAP            $DISK.FASTSORT.SWAPFILE
    MODE            MINSPACE
    SUBSORTS        (=SUBSORTA,=SUBSORTB,=SUBSORTC,=SUBSORTD)
```

You can also use the ALTER DEFINE command to modify a SUBSORT DEFINE name of an existing SORT DEFINE. The following command removes =SUBSORTA:

```
ALTER DEFINE =distributor_collector,          &
      SUBSORTS (=subsortb, =subsortc, =subsortd)
```

```
INFO DEFINE =distributor_collector, DETAIL
    Define Name     =DISTRIBUTOR_COLLECTOR DEFINE
    CLASS           SORT
    SWAP            $DISK.FASTSORT.SWAPFILE
    MODE            MINSPACE
    SUBSORTS        (=SUBSORTB,=SUBSORTC,=SUBSORTD)
```

To delete an attribute from the working attribute set before you create a new DEFINE, use the RESET DEFINE command. The following example deletes all previously specified SUBSORT DEFINE names for the =DISTRIBUTOR_COLLECTOR DEFINE.

```
RESET DEFINE SUBSORTS
```

Deleting a DEFINE

Use the DELETE DEFINE command to delete a DEFINE. In the following example, the first command deletes =SUBSORTB, while the second command deletes all DEFINEs (including DEFINEs other class SORT DEFINEs). A double asterisk (**) or an equal sign and asterisk (=*) in the DELETE DEFINE command specifies all DEFINEs.

```
DELETE DEFINE =SUBSORTB
DELETE DEFINE **
```

Using DEFINEs With Interactive FastSort

After you create a SORT DEFINE, you can use it with interactive FastSort by specifying the DEFINE name in the FastSort RUN command. FastSort reads the attributes from the SORT DEFINE and then uses them as parameters for the sort or merge run. If you specify a DEFINE other than class SORT in the RUN command, FastSort returns an error message.

The following FastSort command file includes a RUN command that uses the =DISTRIBUTOR_COLLECTOR SORT DEFINE:

```
FROM infile
TO outfile
ASC 1:10
RUN, DEFINE =distributor_collector
...
```

The *Guardian User’s Guide* also provides information and examples for creating and using TACL DEFINEs interactively.

Creating and Using DEFINEs Programmatically

You create and use a SORT or SUBSORT DEFINE (and other TACL DEFINEs as well) programmatically using the system procedures shown in the table below. The operating system places the DEFINEs in the process file segment (PFS) of your application.

Procedure	**Description**
DEFINEADD	Creates a DEFINE for the user from the working attribute set.
DEFINEDELETE	Deletes a specific DEFINE for the user.
DEFINEDELETEALL	Deletes all DEFINEs for the user.
DEFINEINFO	Returns information about a DEFINE.
DEFINEMODE	Sets the DEFINE mode (DEFMODE variable) for the user process.
DEFINENEXTNAME	Returns the name of the DEFINE that follows the specified DEFINE.
DEFINEPOOL	Designates part of the user stack or extended data segment as a pool.
DEFINERADATTR	Returns the current value of a specified DEFINE attribute.
DEFINERESTORE	Restores a saved DEFINE from a user-specified buffer for active use.
DEFINERESTOREWORK2	Restores the working set from a background set.
DEFINESAVE	Copies an active DEFINE to a user-specified buffer.
DEFINESAVEWORK[2]	Saves a first or second DEFINE working set in the background set.
DEFINESETATTR	Modifies an attribute in the working set.
DEFINESETLIKE	Initializes the working set with values from an existing DEFINE.
DEFINEVALIDATEWORK	Checks the working set for consistency.
CHECKDEFINE	Checkpoints a DEFINE to a backup process.

For a detailed description, including the syntax, of these procedures, see the *Guardian Procedure Calls Reference Manual*.

Creating and Modifying DEFINEs Programmatically

To use TACL DEFINEs programmatically, you must first set the DEFMODE variable to ON for your application by using one of the following methods:

- Before you run your application, issue the SET DEFMODE ON command from your TACL process.
- From your application, call the DEFINEMODE procedure with the *`new^value`* parameter set to 1.

After you enable DEFINEs, you use the DEFINESETATTR procedure to set the values of attributes, including the CLASS attribute, in the working attribute set. (You can also issue the SET DEFINE CLASS SORT command from your TACL process before your run your application.) After you have set the necessary attributes, you use the DEFINEADD procedure to name the DEFINE and add it to your application’s PFS.

The following TAL example shows the programmatic use of a SORT DEFINE named =SORT^DEFINE. The CONVERT^INT^TO^STRING and ERROR^RECOVERY procedures not shown in this example are user-written procedures. CONVERT^INT^TO^STRING converts an integer value to a character string for use in the DEFINESETATTR procedure call. ERROR^RECOVERY processes any errors that occur in the system procedure calls.

```
STRING .sort^define^name[0:23],
       .attribute^name[0:15],
       .attribute^value[0:15];

INT attribute^length,
    sortprog^cpu^number,
    error;

... ! Enter the SORT DEFINE attribute values from a terminal.

sort^define^name ':=' "=sort^define            ";
attribute^name   ':=' "CLASS           ";
attribute^value  ':=' "SORT            ";
attribute^length  := 4;
error := DEFINESETATTR (attribute^name,
                        attribute^value,
                        attribute^length);
IF error <> 0 THEN CALL error^recovery;

attribute^name ':=' "CPU             ";
error := convert^int^to^string (sortprog^cpu^number,
                                attribute^value,
                                attribute^length);
IF error <> 0 THEN CALL error^recovery;

error := DEFINESETATTR (attribute^name,
                        attribute^value,
                        attribute^length);
IF error <> 0 THEN CALL error^recovery;

... ! Set any other SORT DEFINE attributes.

error := DEFINEADD (sort^define^name);
IF error <> 0 THEN CALL error^recovery;
...
```

Using DEFINEs With Programmatic FastSort

To use a SORT DEFINE other than =_SORT_DEFAULTS in a program, specify the DEFINE name in the SORTBUILDPARM procedure described in Section 5, Using FastSort System Procedures. If you omit the DEFINE name parameter in SORTBUILDPARM, or if you specify a name of all blanks, FastSort does not check for a SORT DEFINE.

The following TAL example uses a SORT DEFINE named =SORT^DEFINE in the SORTBUILDPARM procedure. The operating system does not check the existence or validity of the SORT DEFINE until the sort operation begins.

```
PROC sort^procedure;
BEGIN
  INT .sort^define^name[0:11] := [ 12 * ["  "]];

...

sort^define^name ':=' ["=sort^define            "];

... ! Set the other SORTBUILDPARM parameters.

status := SORTBUILDPARM (sortblock,         ! Control block
                         cpu-mask,
                         not-cpu-mask,
                         buffer,
                         buffer2,
                         buffer^length,
                         flags,
                         sort^define^name); ! DEFINE name
...
```

For more information about using DEFINEs programmatically, see *Guardian Programmer’s Guide*.

Creating and Using the =_SORT_DEFAULTS DEFINE

In a =_SORT_DEFAULTS DEFINE you can specify FastSort parameters for applications that otherwise cannot set the parameters. For example, if a SQL/MP query uses FastSort to sort rows from a table, FastSort uses attributes from the =_SORT_DEFAULTS DEFINE if it exists and DEFMODE is set to ON. The =_SORT_DEFAULTS DEFINE is the only DEFINE you can use to configure a sort from SQL/MP. Other SORT and SUBSORT DEFINEs do not affect SQL/MP sorts.

HP recommends that you use the =_SORT_DEFAULTS DEFINE only for serial sort operations. If you use the =_SORT_DEFAULTS DEFINE to configure a parallel sort operation, follow the guidelines in Selecting a Scratch Volume for Parallel Sorts on page 9-7 and Specifying a Swap File for Parallel Sorts on page 9-10 to avoid sort failure.

Although the =_SORT_DEFAULTS DEFINE name is reserved for use as the default FastSort DEFINE, you create it just as you create other DEFINEs. The following example creates a =_SORT_DEFAULTS DEFINE and displays its attributes. The current attribute set is adopted from the working attribute set.

```
ADD DEFINE =_SORT_DEFAULTS, CLASS SORT
INFO DEFINE =_SORT_DEFAULTS, DETAIL
```

```
    Define Name         =_SORT_DEFAULTS
    CLASS               SORT
```

The following ADD DEFINE command creates the =_SORT_DEFAULTS DEFINE and sets the SCRATCH, SWAP, and CPU attributes:

```
ADD DEFINE =_SORT_DEFAULTS, CLASS SORT,         &
    SCRATCH  $disk.scratch.file                 &
    SWAP  $disk.swap.file                       &
    CPU 8
```

You can change the settings of the current attributes of the =_SORT_DEFAULTS DEFINE using the ALTER DEFINE command. You can then use the INFO DEFINE command to display the new attribute values.

```
ALTER DEFINE =_SORT_DEFAULTS, SCRATCH $disk
ALTER DEFINE =_SORT_DEFAULTS, PRI 150
INFO DEFINE =_SORT_DEFAULTS, DETAIL
    Define Name         =_SORT_DEFAULTS
    CLASS               SORT
    SCRATCH             $DISK
    PRI                 150
```

To use the =_SORT_DEFAULTS DEFINE with interactive FastSort, you do not need to specify it the RUN command, as shown in the following example.

```
> SORT
FastSort - T9620D30 - (31OCT94)
COPYRIGHT TANDEM COMPUTERS INCORPORATED 1991 - 1994
<FROM fruit
<ASC 1:10
<RUN
apple
banana
grape
grapefruit
lemon
orange
pear
watermelon
          8  RECORDS               132  MAX RECORD SIZE
      00:02  ELAPSED TIME           63  BUFFER PAGES
      00:00  I/O WAIT TIME           0  INITIAL RUNS
         19  COMPARES                0  MERGE ORDER
          0  SCRATCH DISK
          0  SCRATCH SEEKS
Errors detected: 0
Warnings detected: 0
```

In the next example, an invalid disk volume name for the =_SORT_DEFAULTS DEFINE SCRATCH attribute causes the FastSort error message (A SCRATCH FILE CANNOT BE OPENED) and file-system error 14 (DEVICE DOES NOT EXIST).

```
SORT
FastSort - T9620D30 - (31OCT94)
COPYRIGHT TANDEM COMPUTERS INCORPORATED 1991 - 1994
```

```
<FROM fruit
<ASC 1:10
<RUN
 *** ERROR ***  A SCRATCH FILE CANNOT BE OPENED.
OPERATING SYSTEM ERROR: 14
SCRATCH FILE: \SYS.$VOLUME
```

To correct the error, use the ALTER DEFINE command to set the SCRATCH attribute to a valid name and then run FastSort again.

```
ALTER DEFINE =_SORT_DEFAULTS, SCRATCH $data

SORT
...
```

Examples of SORT and SUBSORT DEFINEs

This subsection contains the following examples:

- A serial sort run using a SORT DEFINE
- A parallel sort run with a SORT DEFINE for a distributor-collector process and two SUBSORT DEFINEs for the subsort processes

Serial Sort Run Example

The first example creates a SORT DEFINE named =SORT_RUN for a serial sort operation. The SET commands set the DEFINE mode (DEFMODE) to ON and the DEFINE CLASS to SORT. The SHOW DEFINE command then displays the available attributes in the working attribute set:

```
SET DEFMODE ON
SET DEFINE CLASS SORT
SHOW DEFINE *
    CLASS              SORT
    SCRATCH
    SCRATCHON
    NOSCRATCHON
    SWAP
    MODE
    CPU
    BLOCK
    PRI
    SEGMENT
    PROGRAM
    CPUS
    NOTCPUS
    SUBSORTS
    VLM
```

Set selected attributes in the working attribute set:

```
SET DEFINE SCRATCH $disk
SET DEFINE SWAP $data
SET DEFINE CPU 5
SET DEFINE MODE AUTOMATIC
SET DEFINE PRI 170
```

Display the current attribute set:

```
SHOW DEFINE *
    CLASS                  SORT
    SCRATCH                $DISK
    SCRATCHON
    NOSCRATCHON
    SWAP                   $DATA
    MODE                   AUTOMATIC
    CPU                    5
    BLOCK
    PRI                    170
    SEGMENT
    PROGRAM
    CPUS
    NOTCPUS
    SUBSORTS
    VLM
```

Create the SORT DEFINE and display the current attribute set for all current DEFINEs. The current attribute set for the =SORT_RUN DEFINE is adopted from the working attribute set.

```
ADD DEFINE =sort_run
INFO DEFINE **, DETAIL
    Define Name            =SORT_RUN
    CLASS                  SORT
    SCRATCH                $DISK
    SWAP                   $DATA
    MODE                   AUTOMATIC
    CPU                    5
    PRI                    170

    Define Name            =_DEFAULTS
    CLASS                  DEFAULTS
    VOLUME                 $DISK.SUBVOL
    SWAP                   $DATA
```

Run FastSort and specify the =SORT_RUN DEFINE in the RUN command:

```
> SORT
FastSort - T9620D30 - (31OCT94)
COPYRIGHT TANDEM COMPUTERS INCORPORATED 1991 - 1994
<FROM fruit
<TO sortout
<ASC 1:10
<RUN, DEFINE =sort_run
...
```

Parallel Sort Run Example

The following example shows a parallel sort run using a SORT DEFINE named =PARALLEL_SORT and SUBSORT DEFINEs named =SUBSORTA and =SUBSORTB. (The input file FRUIT contains only 8 records; however, an actual input file would be much larger to require a parallel sort operation.)

```
SET DEFMODE ON
ADD DEFINE =parallel_sort, CLASS SORT,      &
    SCRATCH $disk.fastsort.scratch,         &
    CPU 5,                                  &
    PRI 145,                                &
    SUBSORTS (=subsorta, =subsortb)

INFO DEFINE =parallel_sort, DETAIL
    Define Name        =parallel_sort
    CLASS              SORT
    SCRATCH            $DISK.FASTSORT.SCRATCH
    CPU                5
    PRI                145
    SUBSORTS           (=SUBSORTA,=SUBSORTB)
```

To create SUBSORT DEFINEs, first set the DEFINE CLASS to SUBSORT. Then use the SHOW DEFINE command to display the available attributes in the working attribute set. The two question marks (??) indicate a required attribute that you must supply.

```
SET DEFINE CLASS SUBSORT
SHOW DEFINE *
    CLASS              SUBSORT
    SCRATCH            ??
    SWAP
    CPU
    PRI
    SEGMENT
    PROGRAM
Current attribute set is incomplete
SET DEFINE SCRATCH $disk
```

Create SUBSORT DEFINEs using the names specified in the ADD DEFINE command for the =PARALLEL_SORT DEFINE, and then display all your current DEFINEs (including the =_DEFAULTS DEFINE):

```
ADD DEFINE =subsorta, CPU 3
ADD DEFINE =subsortb, CPU 6
INFO DEFINE **, DETAIL
    Define Name        PARALLEL_SORT
    CLASS              SORT
    SCRATCH            $DISK.FASTSORT.SCRATCH
    CPU                5
    PRI                145
    SUBSORTS           (=SUBSORTA,=SUBSORTB)

    Define Name        =SUBSORTA
    CLASS              SUBSORT
    SCRATCH            $DISK
    CPU                 3

    Define Name         =SUBSORTB
    CLASS               SUBSORT
    SCRATCH             $DISK
    CPU                 6

    Define Name         =_DEFAULTS
    CLASS               DEFAULTS
    VOLUME              $DISK.SUBVOL
    SWAP                $DATA
```

Run the =PARALLEL_SORT DEFINE with interactive FastSort:

```
SORT
FastSort - T9620D30 - (31OCT94)
COPYRIGHT TANDEM COMPUTERS INCORPORATED 1991 - 1994
<FROM fruit
<ASC 1:10
<RUN, DEFINE =parallel_sort
apple
banana
grape
grapefruit
lemon
orange
pear
watermelon
          8  RECORDS                132  MAX RECORD SIZE
      00:07  ELAPSED TIME            63  BUFFER PAGES
      00:03  I/O WAIT TIME            0  INITIAL RUNS
         27  COMPARES                15  MERGE ORDER
          0  SCRATCH DISK             0  FIRST MERGE
          0  SCRATCH SEEKS
Errors detected: 0
Warnings detected: 0
```

8 Sorting From NonStop SQL/MP

Under certain circumstances, SQL/MP invokes FastSort in a manner that is transparent to the user. SQL/MP invokes FastSort when you do any of the following:

- Specify ordering or grouping options in an SQL query statement
- Execute a query that results in a sort merge join operation
- Use a CREATE INDEX or LOAD statement to load data in parallel

Because SQL/MP automatically invokes FastSort, this section describes how a sort operation is implemented. This section also contains guidelines on how to minimize SQL sorts and configure your FastSort environment.

How SQL/MP Implements a Sort

The SQL optimizer analyzes each SQL statement and determines if a sort is needed. If needed, SQL/MP implements the sort in one of two ways.

In-memory Sorts

An in-memory sort is the fastest type of sort operation because it requires no SORTPROG process or scratch files. FastSort can sort records within the executor’s extended memory segment if all of the following conditions apply:

- The data to sort is less than 4 MB
- The number of rows to sort is less than 32,768
- The number of columns to sort is less than 63

For the serial portion of a parallel plan, the optimizer can also choose an in-memory sort if all of the following conditions apply:

- The master executor server process (ESP) uses an in-memory sort to execute an ORDER BY clause.
- A GROUP BY clause precedes the ORDER BY clause.
- SQL/MP uses hash grouping to execute the GROUP BY clause.

Note. The optimizer does not choose an in-memory sort if the table to sort is the inner table of a sort merge join operation.

External Physical Sorts

SQL/MP uses an external physical sort if the memory segment is too small to hold all of the rows. An external physical sort is one of the following:

- An external FastSort process (SORTPROG), if the SQL optimizer estimates that the data to sort might exceed 4 MB, or there are fewer than 32,768 rows or fewer than 63 columns
- A sort performed by a series of inserts into a temporary key-sequenced table, if both of the following conditions apply:
    - The table contains more than 500 rows and more than 63 columns
    - The total key length is less than 255 bytes

Note. If the total key length is greater than 255 bytes SORTPROG returns an error.

Configuring Your SQL/MP Sort Environment

Depending on the SQL operation you perform, you can configure your FastSort environment for SQL/MP in three ways:

- Using the =_SORT_DEFAULTS DEFINE
- Using a configuration file for a parallel index load
- Using LOAD command options

Setting Up a =_SORT_DEFAULTS DEFINE

The =_SORT_DEFAULTS DEFINE is the DEFINE you use to configure sorts from SQL/MP. While configuration file and LOAD options only affect loading data, the =_SORT_DEFAULTS DEFINE affects all SQL operations that invoke FastSort. If you do not specify a configuration file or LOAD command options, SQL/MP uses values in your =_SORT_DEFAULTS DEFINE for the load operation.

Note. Before you can use a =SORT_DEFAULTS DEFINE, you must enable DEFINEs for your TACL session. To enable DEFINEs, execute the SET DEFMODE ON command from either your TACL or SQLCI prompt. For more information about this command, see Section 7, Using SORT and SUBSORT DEFINEs.

You can create or modify a =_SORT_DEFAULTS DEFINE directly from your SQLCI prompt. The syntax for creating a =_SORT_DEFAULTS DEFINE is:

```
ADD DEFINE =_SORT_DEFAULTS, CLASS SORT
    [, BLOCK block-size                      ]
    [, CPU cpu-number                        ]
    [, CPUS subsort-cpu-list                 ]
    [, MODE mode-type                        ]
    [, NOSCRATCHON  ( volume-list )          ]
    [, NOTCPUS cpu-list-not-subsort          ]
    [, PRI process-priority                  ]
    [, PROGRAM file                          ]
    [, SCRATCH file                          ]
    [, SCRATCHON ( volume-list )             ]
    [, SEGMENT extended-segment-size         ]
    [, SUBSORTS define-list                  ]
    [, SWAP file-name                        ]
    [, VLM { ON | OFF }                      ]
```

All TACL DEFINE commands, such as SET DEFINE and ALTER DEFINE, are valid for the =_SORT_DEFAULTS DEFINE. You can also name and set any SORT or SUBSORT DEFINE attribute in a =_SORT_DEFAULTS DEFINE.

You must specify CLASS SORT for the =_SORT_DEFAULTS DEFINE. You can specify CLASS SORT either in the ADD DEFINE command or through the working attribute set. To learn how to use DEFINE commands and SORT and SUBSORT DEFINE attributes, see Section 7, Using SORT and SUBSORT DEFINEs.

The example below creates a =_SORT_DEFAULTS DEFINE and specifies values for the SCRATCH, SWAP, CPU, AND PRI attributes:

```
ADD DEFINE =_SORT_DEFAULTS, CLASS SORT,
         SCRATCH $DATA, SWAP $DATA2, CPU 3, PRI 100
```

At run time, if the scratch and swap files you specify in a =_SORT_DEFAULTS DEFINE do not exist, FastSort automatically creates these files and sets MAXEXTENTS to 160. If you manually create scratch and swap files, size them according to the number of records to sort and extended memory segment size, respectively. For more information about scratch and swap files, see Section 9, Optimizing Sort Performance.

Guidelines for =_SORT_DEFAULTS DEFINE Attributes

To optimize sort performance for SQL/MP, HP recommends you follow these guidelines for SORT and SUBSORT attributes in your =_SORT_DEFAULTS DEFINE:

Attribute	Recommended value
VLM	OFF ON for nonparallel LOAD operations
PRI	180 for high priority users and queries 160 for most users and queries 80 for routine load operations and queries
PROGRAM	A local file
CPU	A processor that is less than 50 to 60 percent busy
SCRATCH	On a local volume other than $SYSTEM. Avoid the volume where the SORTPROG is running. Avoid using volumes on mirrored disks. An empty volume is best.
SWAP	On a local volume other than $SYSTEM and other than the scratch volume

△ **Caution.** For any parallel query, parallel CREATE INDEX operation, or parallel load operation, specify only volume names for the SCRATCH and SWAP attributes in a =_SORT_DEFAULTS DEFINE. Do not specify fully qualified file names for these attributes. If you specify fully qualified scratch and swap file names for a parallel sort operation, processor and disk space contention problems can result.

These attributes are fully described in Section 7, Using SORT and SUBSORT DEFINEs. Check with your system manager to learn how resources are allocated on your node. For more information on how to allocate sort workspace, see Section 9, Optimizing Sort Performance.

Ordering and Grouping Query Results

This subsection describes situations in which your query causes SQL/MP to invoke FastSort, and how to structure a query to use logic built into the SQL optimizer. For more information about the SQL clauses mentioned in this subsection, see the *NonStop SQL/MP Query Guide*.

In general, SQL/MP uses FastSort to order and group query results. SQL/MP automatically invokes FastSort in certain cases when you specify an ORDER BY clause in a query statement and SQL/MP retrieves data from the base table. SQL/MP also uses FastSort when you specify one of the following in a query statement:

- GROUP BY
- DISTINCT

- UNION (without the ALL option)

and the specified columns do not match a prefix of the index columns.

Note. NonStop SQL/MP does not invoke FastSort if the optimizer chooses a query plan that reads the base table by primary key value.

If you specify more than one SQL ordering or grouping clause in a query, you can often structure the query to avoid duplicate sorts. For queries in which the optimizer does not choose a parallel execution plan, you should also use a =_SORT_DEFAULTS DEFINE to optimize performance. For more information about setting up a =_SORT_DEFAULTS DEFINE, see Configuring Your SQL/MP Sort Environment on page 8-2.

Optimizing SQL Clause Combinations

The SQL/MP optimizer attempts to minimize sort operations. However, certain combinations of SQL clauses can still cause unnecessary or duplicate sorts. The following examples show how to structure SQL statements to minimize unnecessary sorts.

Specifying ORDER BY With GROUP BY

You can order and group query results in a single sort when the following occurs:

- The ORDER BY list is a subset of the GROUP BY list

  For example, only one sort is necessary for the following query:

  ```
  SELECT ATLANTA, BOSTON, CHICAGO, DALLAS FROM SALES
     GROUP BY ATLANTA, BOSTON, CHICAGO, DALLAS
     ORDER BY BOSTON, CHICAGO DESC ;
  ```

  A single sort groups and orders the results of this query. In this case, SQL/MP sorts on (BOSTON, CHICAGO, ATLANTA, DALLAS).

- The GROUP BY list contains *n* items, which are also the first *n* items of the ORDER BY list, as in the following query:

  ```
  SELECT ATLANTA, BOSTON, CHICAGO, COUNT(*), SUM(ATLANTA)
  FROM SALES
     GROUP BY ATLANTA, BOSTON, CHICAGO
     ORDER BY 1, 2 DESC, 3, 5, 4 ;
  ```

  A single sort also groups and orders the results of this query. In this case, SQL/MP sorts on (ATLANTA, BOSTON, CHICAGO).

Specifying GROUP BY With DISTINCT

You can group results and eliminate duplicate rows in a single sort when:

- The GROUP BY list is a subset of the SELECT DISTINCT list, as in the following query:

  ```
  SELECT DISTINCT COUNT(*), BOSTON, BOSTON-DALLAS, DALLAS
  FROM SALES
     GROUP BY BOSTON, DALLAS ;
  ```

  A single sort on (BOSTON, DALLAS) groups the query results. Because each (BOSTON, DALLAS) value is unique after grouping, each (BOSTON, DALLAS, BOSTON-DALLAS, COUNT(*)) value is also unique.

- The SELECT DISTINCT list is a subset of the GROUP BY list, there are no expressions in the SELECT list, and no aggregates in a HAVING clause, as in the following query:

  ```
  SELECT DISTINCT ATLANTA, CHICAGO FROM SALES
      GROUP BY ATLANTA, BOSTON, CHICAGO ;
  ```

  In this case, only a single sort is required because the GROUP BY clause is unnecessary. Because BOSTON is not in the SELECT list and no aggregates or HAVING clauses rely on the full grouping, there is no need to group by BOSTON.

  To build this logic into your query and avoid the unnecessary sort, add the DISTINCT column to the GROUP BY list.

Specifying ORDER BY With DISTINCT

You can order query results and eliminate duplicate rows in a single sort if the ORDER BY list is a subset of the DISTINCT list, as in the following query:

```
SELECT DISTINCT ATLANTA, BOSTON, CHICAGO, DALLAS FROM SALES
   GROUP BY ATLANTA, BOSTON DESC ;
```

In this case, a single sort on (ATLANTA, BOSTON DESC, CHICAGO, DALLAS) orders results and eliminates duplicate rows. The position and sorting order, ascending or descending, of ATLANTA and BOSTON must match the index used for the sort. However, CHICAGO and DALLAS can occur in any order after ATLANTA and BOSTON, and in either ascending or descending order.

Using a Sort Merge Join

A join operation combines data from two tables or views. The sort merge join is one of four join methods available to the SQL/MP optimizer. The optimizer evaluates query cost and decides which type of join to perform.

For the optimizer to choose a sort merge join, these conditions must exist:

- The joining columns of outer and inner tables must be in ascending or descending order
- The query must be an equijoin query

During a sort merge join, FastSort always sorts the data from the inner table and stores it in a temporary entry-sequenced table. If the outer table is not already sorted on the joining column, FastSort also sorts the outer table data and stores it in a second temporary entry-sequenced table. The two temporary tables are then merged to form the sort merge join result.

By default, FastSort creates these temporary tables on the default swap volume. To avoid disk space contention, move the swap file to a volume other than the default. For information on how to specify swap file location, see Section 9, Optimizing Sort Performance

For more information about sort merge joins and equijoin queries, see *SQL/MP Query Guide*.

Loading Data

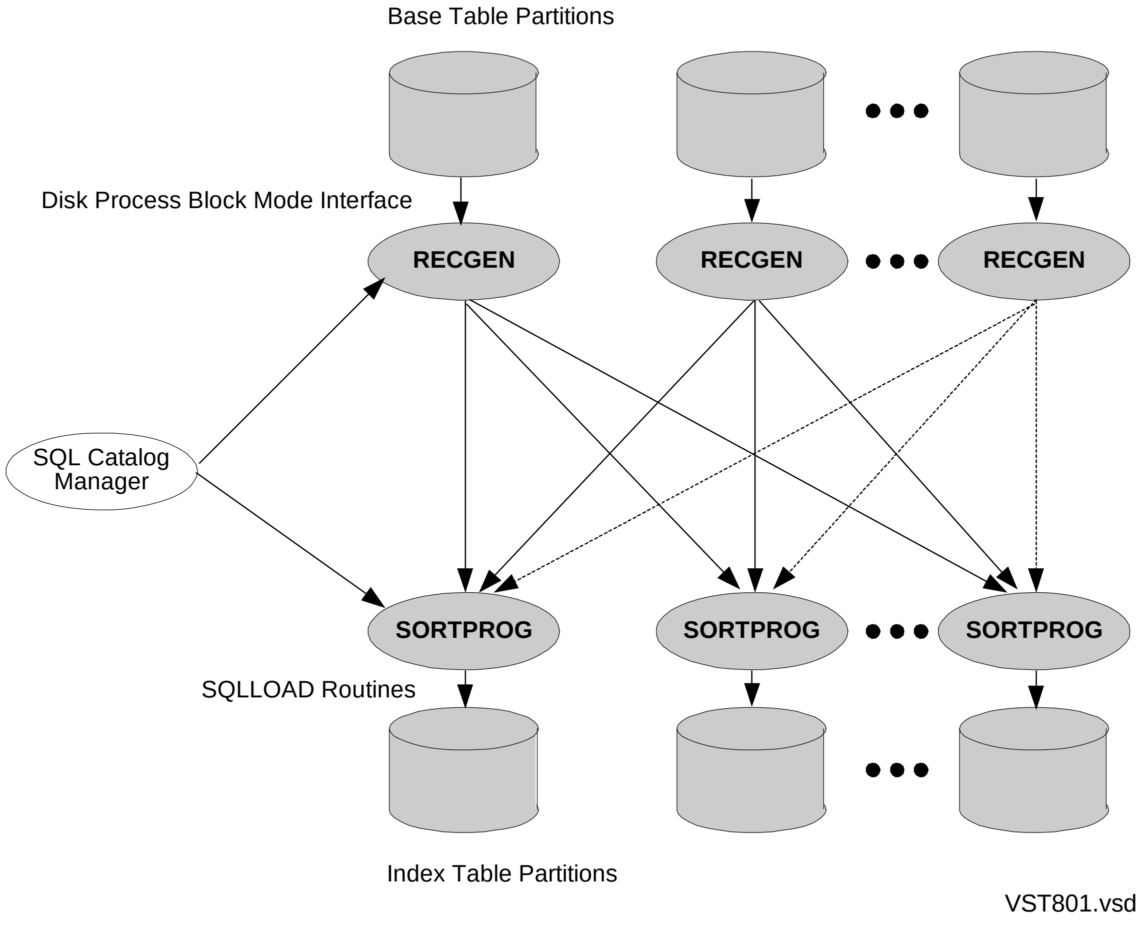
When you execute CREATE INDEX or LOAD and the source table contains data, SQL/MP uses FastSort to help process the data under these circumstances:

- CREATE INDEX with PARALLEL EXECUTION ON
- LOAD with PARALLEL EXECUTION ON
- LOAD without the SORTED option if the target table is key-sequenced

If the target table is partitioned, you can specify PARALLEL EXECUTION ON to load partitions in parallel. SQL/MP starts a record generator (RECGEN) process for each partition of the table and a sort process (SORTPROG) for each partition of the index. Record generator processes read the base table rows. Sort processes sort the generated rows and write them to the index.

Figure 8-1 on page 8-8 shows the interaction between the SQL/MP catalog manager and RECGEN and SORTPROG processes when you load data in parallel. If neither base table nor index is partitioned, SQL/MP uses only one RECGEN process and one SORTPROG process.

Figure 8-1. Parallel Loading Data Into a Partitioned Index Table



Because parallel processing uses more concurrent CPU cycles and disk processes than serial processing, loading data in parallel could temporarily monopolize system resources. Try to schedule other system tasks accordingly.

The default location of a RECGEN process is $SYSTEM.SYS*nn*.RECGEN, where *nn* is a two-digit number assigned by HP. To specify a different location, use the =_SQL_RECGEN_*node* DEFINE. You must have super ID authority on the specified node to move a RECGEN process. For more information about this DEFINE, see *SQL/MP Reference Manual*.

Configuring a CREATE INDEX Statement

When you create an index on a base table and do not specify PARALLEL EXECUTION ON, you can use either the default configuration described in this subsection or a =_SORT_DEFAULTS DEFINE. For more information on how to set up a =_SORT_DEFAULTS DEFINE, see Configuring Your SQL/MP Sort Environment on page 8-2. For detailed information about SORT DEFINEs and DEFINE attributes, see Section 7, Using SORT and SUBSORT DEFINEs.

When you create a partitioned index on a base table and specify PARALLEL EXECUTION ON, you can use either the default configuration or a custom configuration file. A configuration file defines attributes of record generator and sort processes. You can specify the name of a configuration file in the PARALLEL EXECUTION clause of a CREATE INDEX statement. If you specify no configuration file, FastSort uses the default configuration.

Using the Default Configuration

If you specify PARALLEL EXECUTION ON and do not specify a configuration file, SQL/MP uses the following defaults:

Default Scratch File Size Formula

Under the default configuration, SQL/MP uses the following formula to estimate scratch file size:

$$3 \times \frac{NumberofBaseTableRecords}{NumberofIndexTablePartitions}$$

NonStop SQL/MP estimates the number of records in the base table by dividing file size by record length.

Default DEFINE Attribute Values

The default configuration for CREATE INDEX includes the following attribute values:

Attribute	Default value
PRI	For both RECGEN and SORTPROG, the priority of the process that creates the index.
CPU	For local partitions, the CPU that runs the primary disk process for that partition. If a partition is remote or a CPU is unavailable, FastSort arbitrarily selects the first CPU, then selects subsequent CPUs in a sequential fashion. Note that multiple RECGENs or SORTPROGs can run in a single CPU.
SCRATCH	FastSort selects a volume for the initial SORTPROG scratch file. RECGEN processes do not use scratch files.
SWAP	For SORTPROGs, the scratch volume if that volume is local and if not, the volume where the SORTPROG is running. For RECGENs, the volume of the partition being read if that partition is local and if not, the swap volume specified in the =_SORT_DEFAULTS DEFINE.

The default configuration is not recommended for base tables with many remote partitions. Note that in this configuration, all record generator processes that read remote partitions swap to the same volume. When multiple processes swap to the same volume, processor and disk space contention problems can result. Use a configuration file to specify a unique swap volume for each remote partition.

Using a Custom Configuration File

When you create an index and specify PARALLEL EXECUTION ON, you can use the CONFIG option to specify a custom configuration file. The configuration file must be an EDIT file. It can describe either a default configuration or an explicit configuration for both record generator and sort processes. Default and explicit configurations are discussed in Assigning Default and Explicit Values on page 8-14.

The values you specify in a configuration file override any values in a =_SORT_DEFAULTS DEFINE. This subsection describes configuration file syntax and contains a sample file.

△ **Caution.** HP recommends that you use only a custom configuration file to configure a parallel CREATE INDEX operation. If you omit the SCRATCH and SWAP attributes in your configuration file and a =_SORT_DEFAULTS DEFINE contains fully qualified file names for these attributes, processor and disk space contention problems can result.

Configuration File Syntax

The two types of statements in a configuration file are COMMENT and CREATE INDEX. Keywords in the configuration file can be in uppercase, lowercase, or mixed-case letters. The maximum length for a configuration file statement is 132 characters. However, due to parser requirements, a line in a configuration file can be at most 80 characters long. Split statements longer than 80 characters into two lines of up to 80 characters each. In this case, insert an ampersand character (&) at the end of the first line to specify that the lines make up a single statement.

Use the COMMENT statement to include descriptive notes in the file. SQL/MP ignores lines that begin with the keyword COMMENT or the characters ==. The syntax for COMMENT is as follows:

```
{ COMMENT comment-text }
{ == comment-text      }
```

The CREATE INDEX statement precedes all configuration information for the parallel load operation. SQL/MP reads only CREATE INDEX statements in the configuration file. The syntax for CREATE INDEX is as follows:

```
           {LOCALONLY}
CREATEINDEX{BASETABLE}{DEFAULT [node-name] default-attr   }
           {INDEX    }{partition attr [,attr ]...         }

where default-attr is:

   [ CPU ( num  [, num ] ... )                            ]
   [ NOSCRATCHON (scratchvol [, scratchvol ]...)          ]
   [ NUMRECS ( number )                                   ]
   [ PRI ( priority )                                     ]
   [ PROGRAM (filename )                                  ]
   [ SCRATCH  (scratchvol [, scratchvol ]...)             ]
   [ SCRATCHON (scratchvol [, scratchvol ]...)            ]
   [ SWAP ( swapvol )                                     ]

and where attr is:

   [ CPU (num )                                           ]
   [ NOSCRATCHON  (scratchvol [, scratchvol ]...)         ]
   [ NUMRECS  ( number )                                  ]
   [ PRI  ( priority )                                    ]
   [ PROGRAM (filename )                                  ]
   [ SCRATCH    scratchvol                                ]
   [ SCRATCHON  (scratchvol [, scratchvol ]...)           ]
   [ SWAP  ( swapvol )                                    ]
```

`LOCALONLY`

directs SQL/MP to run the SORTPROG and RECGEN processes only on the local node. For performance reasons, the default location is remote. If there is no remote node, then SORTPROG and RECGEN run locally. Use this option only on systems with remote nodes to preserve software behavior available in previous RVUs. If you specify LOCALONLY, it must be the first CREATE INDEX statement in the configuration file.

`BASETABLE`

applies the attributes you specify to the record generator processes that read the base table.

`INDEX`

applies the attributes you specify to the sort processes that write to index partitions.

`DEFAULT [`*`node-name`*`]` *`default-attr`*

specifies an attribute-value pair for partitions on a node for which no value has been explicitly specified. If you omit *`node-name`*, SQL/MP applies the DEFAULT statement to the node where the parallel index load is initiated. For more information on how to use this option, see Assigning Default and Explicit Values on page 8-14.

`partition`

specifies the name of the volume that contains the partition to which the specified attributes apply. You can include a node name, such as:

```
$myvol
\nwreg.$sales1
```

The default is the local node.

`CPU (` *`num`* `[,` *`num`* `] ... )`

is valid only if you specify INDEX or BASETABLE. CPU specifies one or more local CPUs for record generator or sort processes. You can specify multiple CPUs only as DEFAULT CPUs.

`NOSCRATCHON (` *`scratchvol`* `[,` *`scratchvol`* `] ...)`

is valid only if you specify INDEX. NOSCRATCHON specifies one or more volumes to be excluded as overflow scratch volumes for the sort process. You can use the NOSCRATCHON option either as a DEFAULT specification or for a certain partition. You cannot specify both SCRATCHON and NOSCRATCHON.

When selecting scratch volumes, FastSort consults this list if you do not use the SCRATCHON option to specify a set of overflow scratch volumes. FastSort does not use $SYSTEM or TMF audit trail volumes for overflow scratch files. Volumes with less than 1 MB of disk space and volumes protected by the Safeguard product are also exempt. You can use the NOSCRATCHON option either as a DEFAULT specification or for a certain partition.

When you specify NOSCRATCHON volumes in a configuration file, the values you specify override any values in a =_SORT_DEFAULTS DEFINE.

You can use the wild-card characters * and ? when you specify *`scratchvol`*. See the description of SCRATCHON in Section 7, Using SORT and SUBSORT DEFINEs for examples of how to use these characters.

△ **Caution.** HP recommends that you use only a custom configuration file to configure a parallel CREATE INDEX operation. If you omit the SCRATCH and SWAP attributes in your configuration file and a =_SORT_DEFAULTS DEFINE contains fully qualified file names for these attributes, processor and disk space contention problems can result.

`NUMRECS ( `*`number`*` )`

is valid only if you specify INDEX. NUMRECS specifies the approximate number of records to load into the index partition. Use NUMRECS if the index is unevenly partitioned across volumes. FastSort uses this number to calculate initial scratch file size.

`PRI ( `*`priority`*` )`

is valid only if you specify INDEX or BASETABLE. PRI specifies the priority at which to run the record generator or sort process.

`PROGRAM (`*`filename`*` )`

specifies the name of a local or remote SORTPROG object file if you also specify BASETABLE. If you specify INDEX, PROGRAM specifies the name of a local or remote RECGEN object file. The associated swap volume must reside on the same node as the object file.

`SCRATCH ( `*`scratchvol`*` [, `*`scratchvol`*` ]...)`

is valid only if you specify INDEX. SCRATCH specifies the name of an initial scratch volume or volumes FastSort can use to sort index records. When you specify scratch volumes in a configuration file, the values you specify override any values in a =_SORT_DEFAULTS DEFINE.

You can specify a list of SCRATCH volumes only in DEFAULT syntax. When you use SCRATCH to list default scratch volumes, FastSort assigns one volume to each sort process in a sequential fashion. If there are more index partitions than volumes available, FastSort reuses volumes on the list until each partition has an initial scratch volume. You can use the SCRATCH option either as a DEFAULT specification or for a certain partition.

The SCRATCH option specifies initial scratch volumes. Use the SCRATCHON option to specify a set of overflow volumes. To direct FastSort to set up an overflow scratch volume pool by excluding certain volumes, use the NOSCRATCHON option.

`SCRATCHON ( `*`scratchvol`*` [, `*`scratchvol`*` ]...)`

is valid only if you specify INDEX. SCRATCHON specifies one or more volumes FastSort can use for overflow scratch files. FastSort uses overflow scratch volumes only if one or more initial volumes becomes full, or if you do not use the SCRATCH option to specify an initial scratch volume. You can use the SCRATCHON option either as a DEFAULT specification or for a certain partition.

You can specify up to 32 scratch volumes, within the maximum line length of 132 characters. You can also use the wild-card characters * and ? when you specify *`scratchvol`*. See the description of SCRATCHON in Section 7, Using SORT and SUBSORT DEFINEs for examples of how to use these characters.

When you specify overflow scratch volumes in a configuration file, the values you specify override any values in a =_SORT_DEFAULTS DEFINE.

You cannot specify both NOSCRATCHON and SCRATCHON. If you do not specify either SCRATCHON or NOSCRATCHON, FastSort considers using any volume, except $SYSTEM and TMF audit trail volumes, for overflow scratch files. Volumes with less than 1 MB of disk space and volumes protected by the Safeguard product are also exempt.

`SWAP (` *`swapvol`* `[,` *`swapvol`* `]...)`

is valid only if you specify INDEX or BASETABLE. SWAP specifies the name of the volume on which to place the extended segment swap file. You can include a node name in *`swapvol`*, as in this example:

```
$myvol
\nwreg.$sales1
```

You can specify multiple swap volumes only as DEFAULT swap volumes.

Assigning Default and Explicit Values

You can specify any attribute in a configuration file as either a default or explicit value for record generator or sort processes. Use the DEFAULT option to specify a default value. Use the partition option to specify an explicit value.

Use default values when you want FastSort to choose from a set of multiple values. For example, you might specify scratch volume names or CPU numbers as default values. You can also use default values to apply a single value, such as number of records to sort or execution priority, to all processes.

Use explicit values when you want FastSort to use particular values for size limitation or performance reasons. For example, if one index partition is much larger than others, you might want to explicitly specify values for that partition.

Default and explicit values are not mutually exclusive. For example, you can explicitly specify scratch volumes and specify a default pool of CPUs for the sort processes, or you might specify a default pool of scratch volumes but assign a particular scratch volume to one partition.

Sample Configuration File

Following is a sample configuration file for loading data from the base table CUSTOMER into partitions on the index AGEINDEX. It includes both default and explicit values.

```
==  Sample configuration file for loading index partitions
==  in parallel.  Creates index AGEINDEX on table CUST, which
==  is partitioned as follows:
==      $DATA1.SALES.CUST
==      $DATA2.SALES.CUST
==      $DATA3.SALES.CUST
==      \NEWYORK.$DATA1.SALES.CUST

==  AGEINDEX is partitioned as follows:
```

```
==      $DATA4.SALES.AGEINDEX
==      $DATA5.SALES.AGEINDEX
==      \NEWYORK.$DATA2.SALES.AGEINDEX
        \NEWYORK.$DATA3.SALES.AGEINDEX

==  Set up a default priority for the RECGEN processes:

CREATEINDEX BASETABLE DEFAULT PRI ( 140 )
CREATEINDEX BASETABLE DEFAULT \NEWYORK PRI ( 140 )

==  Set up a default pool of scratch files for the sort
==  processes.

CREATEINDEX INDEX DEFAULT SCRATCH ($TEMP1, $TEMP2, $TEMP3)
CREATEINDEX INDEX DEFAULT \NEWYORK SCRATCH ($TEMP4)

==  Request that overflow scratch files avoid certain disks--
==  those specified plus $SYSTEM and TMF audit trail disks.

CREATEINDEX DEFAULT NOSCRATCHON ($SYS*,$WORK*)

==  Request that overflow scratch files use specific disks
==  on the remote node

CREATEINDEX INDEX DEFAULT \NEWYORK SCRATCHON ($TEMP*)

==  Request that the $data3 sort process use $TEMP7 for
==  scratch files.

CREATEINDEX \NEWYORK.$DATA3 SCRATCH ($TEMP7)


==  End of configuration file.
```

Loading Multiple Indexes

PARALLEL EXECUTION ON applies to only one partitioned index at a time. If the base table has more than one partitioned index, the partitions of the first index are loaded first. After the first index is loaded, the partitions of the second index are loaded in parallel, and so on.

You can use the FOR *`index-name`* clause of the CONFIG option to specify a separate configuration file for each index. If you omit this clause, the configuration file applies to all indexes on the base table. If you specify at least one index in the FOR clause, SQL/MP parallel loads the partitions of any index not specified with the default configuration values. For more information about CONFIG option syntax, see LOAD entry in the *SQL/MP Reference Manual*.

Configuring a LOAD Statement

LOAD is a SQLCI utility you use to load data. LOAD can transfer data from an SQL/MP table or a disk file into either an SQL/MP table and its indexes or an Enscribe structured disk file. LOAD overwrites existing data in the target table or file.

△ **Caution.** To use LOAD you must turn off auditing for the table being loaded. This action invalidates TMF online dumps of the table and its indexes. To ensure TMF rollforward protection for the table and its indexes, make new online dumps of all table and index partitions. If you load only partitions rather than an entire table, turn off auditing and make new online dumps for only the partitions being loaded.

When you execute a LOAD statement from SQLCI, you invoke FastSort if data is unsorted and the target table is key-sequenced, or if PARALLEL EXECUTION is set to ON. This subsection discusses only the LOAD options that affect sort operations. For a full description of LOAD statement syntax, see the *SQL/MP Reference Manual.*

The LOAD options that affect sort operations are:

- SORTED
- MAX
- SCRATCH

These options are only valid for loading key-sequenced files and tables.

`SORTED`

specifies that input records are already sorted in the key-field order of the output file and are not to be resorted. If you omit the SORTED option and the target file is key-sequenced, FastSort sorts the records before LOAD writes data to the output file.

`MAX` *`num-records`*

specifies the number of input records. The range is between 0 and 2,147,483,647. LOAD uses *`num-records`* to determine file and extent size for the initial scratch file. If you specify the SORTED option, you can omit the MAX option.

When you specify *`num-records`*, try to overestimate. If you underestimate the number of records, the sort can be significantly slower. If you overestimate, the cost is small.

The default value for MAX is 50,000 records unless a =_SORT_DEFAULTS DEFINE with VLM ON is in effect. When VLM is on, the default is 1,000,000 records. For more information about that option, see Using VLM on page 9-10.

MAX is not valid for loading indexes. When you load an index, LOAD uses the size of the base table to estimate the number of input records and ignores any value you specify for MAX.

`SCRATCH` *`scratch-file`*

identifies an initial scratch file or volume. For nonparallel load operations, specify the name of either a disk file or volume for *`scratch-file`*. For parallel load operations, specify only a volume name.

If you omit the SCRATCH option, FastSort creates an initial scratch file on a suitable volume unless a =_SORT_DEFAULTS DEFINE that specifies a different initial scratch file or volume is in effect.

When loading a large table, you can use a partitioned scratch file to manage scratch space. Use the FUP CREATE command to create the partitioned file. Then specify the file to FastSort in the SCRATCH option or your =_SORT_DEFAULTS

DEFINE. For more information about partitioned scratch files, see Using a Partitioned Scratch File on page 9-8.

If you specify the SORTED option, you can omit the SCRATCH option.

Loading Large Tables

Use the following sort workspace guidelines to load data into a large table.

Setting MAX Number of Records

LOAD uses the MAX parameter to estimate file and extent size for an initial scratch file. By default, FastSort creates an initial scratch file large enough for only 50,000 records. If VLM is on, the default MAX value is 1,000,000 records.

To ensure efficient use of sort workspace, specify an accurate value for MAX in the LOAD command. To estimate the number of records in the base table, divide file size by record length.

Using VLM With LOAD

VLM shortens the elapsed time of most nonparallel load operations. To use VLM with the SQLCI LOAD command, set VLM ON in a =_SORT_DEFAULTS DEFINE. For more information about VLM, see Using VLM on page 9-10. To learn how to set up a =_SORT_DEFAULTS DEFINE, see Configuring Your SQL/MP Sort Environment on page 8-2.

Do not use VLM for parallel load operations.

Resizing Primary Extent

Large extents can cause problems with sort workspace when you load data from a table into an index.

For large tables, space on the destination disk might be too fragmented to hold the table or index extents. In this case, SORTPROG returns error 29 (A WRITE HAS FAILED TO THE TO FILE) and the load operation fails. Before you load data into a large table, ensure that table extent sizes fit on the destination disk. If extents are too large for the disk, re-create the index and specify a smaller extent size.

Specifying a Partitioned Scratch File

If you load data into a large table, FastSort might require an initial scratch file that is too large to fit on one disk. To estimate initial scratch file size for an SQL/MP load operation, use the formula in Using the Default Configuration on page 8-9.

If not enough continuous disk space exists on your node for an initial scratch file, you can create and use a partitioned scratch file. While the maximum size of a nonpartitioned scratch file is 1 TB if it is created by the user and up to 2 GB otherwise, a partitioned scratch file can be greater than 1 TB. For more information, see Using a Partitioned Scratch File on page 9-8.

9 Optimizing Sort Performance

Factors that affect FastSort performance include environmental options, sort workspace, and system resources. The total elapsed time for a sort operation also depends on whether you automate routine tasks, such as setting up DEFINEs. This section helps you understand FastSort software behavior and requirements. It contains a discussion of scratch and swap files, VLM, and other factors that affect sort performance.

This section mentions utilities and features that help analyze or increase performance. These utilities and features are part of the NonStop Kernel.

Managing Sort Workspace

Most sort failures are caused by insufficient workspace. FastSort requires scratch files, swap files, and memory to sort records. This subsection describes how FastSort allocates space for sort operations. It also suggests ways to control and modify FastSort workspace decisions.

Using Scratch Files

A scratch file is a temporary work file for FastSort. For input files that are too large to sort in memory, FastSort uses one or more scratch files to temporarily store groups of records called runs. You can specify a scratch file in:

- The RUN command
- The SORTMERGESTART procedure
- A SORT DEFINE
    - SCRATCH attribute
    - SCRATCHON attribute
    - NOSCRATCHON attribute
- The SCRATCH attribute of a SUBSORT DEFINE
- The =_SORT_DEFAULTS DEFINE
- A configuration file for parallel index loading
- The LOAD command

The scratch file you specify can already exist. If the file does not exist, FastSort automatically creates it. If the initial scratch file becomes full, FastSort automatically selects a suitable volume and creates overflow scratch files. FastSort can use up to 32 scratch files on up to 32 disk volumes to store intermediate runs. The SORTPROG process sorts each run and then merges the records into the output file.

Manually Creating a Scratch File

You can use the FUP CREATE command to manually create an unstructured scratch file. You can also programmatically create a scratch file with the CREATE system procedure. When you manually create a scratch file, you can:

- Allocate scratch space before the sort operation begins
- Closely control the amount and location of disk space SORTPROG uses

At run time, if an initial scratch file already exists and is unstructured, FastSort uses the existing file. If you manually create your own scratch file, use the following formula to calculate scratch file size:

$$(\mathtt{output}-\mathtt{record}-\mathtt{length}+6\mathtt{bytes})\times\mathtt{input}-\mathtt{record}-\mathtt{count}$$

This formula is approximate, and includes 6 bytes per record for overhead. It does not include scratch block overhead for header information or variations in block size. For a partitioned scratch file, calculate *`input-record-count`* for each partition. For a permutation or key sort, *`output-record-length`* is the total length of all keys. For a record sort, *`output-record-length`* matches the input record length. If you are sorting or merging in parallel, divide file size by the number of subsort processes.

If FastSort creates the scratch file, it sets MAXEXTENTS to 978 extents. If a scratch file reaches MAXEXTENTS, FastSort automatically enlarges the file, if possible. The maximum size of each scratch file extent is 4 KB, or 2048 pages.

If FastSort cannot enlarge the file, SORTPROG tries to create an overflow scratch file on the current volume. If there is insufficient overflow space on the current volume, SORTPROG tries to create an overflow scratch file on a suitable volume. If there is insufficient overflow scratch space on your node, SORTPROG returns FastSort error 30 (A WRITE HAS FAILED TO A SCRATCH FILE) and stops.

Having FastSort Create a Scratch File

If no scratch file exists when the sort or merge run starts, SORTPROG creates an initial scratch file for you. SORTPROG uses a formula like the one described in Manually Creating a Scratch File on page 9-2 to calculate file size. If the initial scratch file becomes full, SORTPROG creates overflow scratch files until the sort or merge run is complete.

For most sort and merge runs, use one of these options to have SORTPROG size and create initial scratch files for you:

- Do not specify a scratch file name. SORTPROG creates an initial scratch file on a volume selected by DEFINEs or volume characteristics.

- Specify a scratch file that does not exist. SORTPROG creates an initial scratch file on a volume selected by DEFINEs or volume characteristics.
- Specify only a volume name. SORTPROG creates an initial scratch file on the specified volume.

Even when SORTPROG creates a scratch file, the file is sometimes too small to hold all of the records. For example, an initial scratch file can be too small if the input record count is smaller than the actual number of input records. In this case, SORTPROG tries to write to a full scratch file and receives file-system error 45 (FILE IS FULL). SORTPROG tries to increase the size of the scratch file by increasing the maximum number of extents until the sort or merge run completes, unless:

- SORTPROG runs out of space on the scratch file disk before the scratch file reaches its maximum limit. SORTPROG then searches for a suitable disk on which to create an overflow scratch file. For more information, see How Volume Characteristics Affect Selection on page 9-5.

  If there is insufficient overflow scratch space, SORTPROG returns FastSort error 30 (A WRITE HAS FAILED TO A SCRATCH FILE) along with file-system error 43 (UNABLE TO OBTAIN DISK SPACE FOR FILE EXTENT) and stops.
- A file-system error other than 21 occurs when SORTPROG is trying to increase the number of extents.

Note. FastSort always purges scratch files after a sort or merge runs completes, unless you sort programmatically and call SORTMERGESTART with *`flags`* parameter bit <12> set to 1.

Initial and Overflow Scratch Volumes

An initial scratch volume is the volume FastSort uses first for scratch files. For example, a volume you specify in the SCRATCH attribute is an initial scratch volume. If you explicitly specify an initial scratch volume, FastSort uses up to 100 percent of available disk space on that volume. If FastSort selects an initial scratch volume, it uses up to 80 percent of available disk space on that volume.

Overflow scratch volumes are volumes FastSort uses as alternate locations for scratch files, if needed. For example, volumes you specify in the SCRATCHON attribute are overflow scratch volumes. If you explicitly specify an overflow scratch volume, FastSort uses up to 100 percent of available disk space on that volume. If FastSort selects an overflow scratch volume, it uses up to 80 percent of available disk space on that volume.

Selecting a Scratch Volume for Serial Sorts

This subsection describes how FastSort selects a scratch volume for serial sorts. For information about subsort scratch files, see Selecting a Scratch Volume for Parallel Sorts on page 9-7.

When a sort operation requires a scratch file, FastSort reads SORT DEFINEs for acceptable scratch volumes. If no scratch file or scratch volume is specified in a

DEFINE, FastSort automatically selects a scratch volume based on volume characteristics. For more information about selection criteria, see How Volume Characteristics Affect Selection on page 9-5.

How DEFINEs Affect Selection

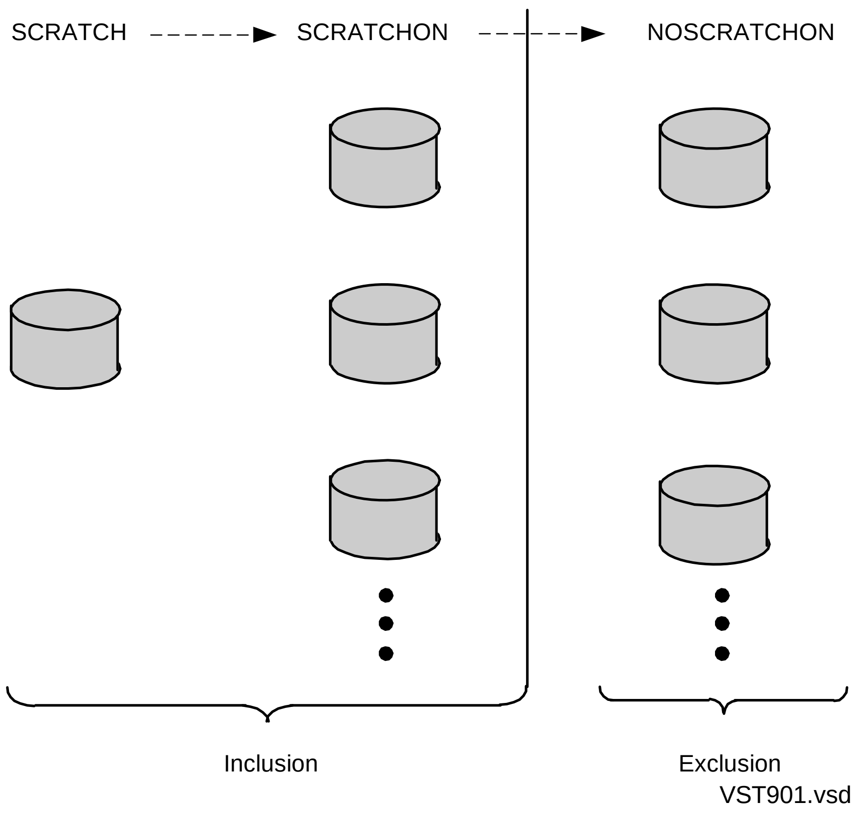
You can specify volumes for FastSort to use or not use for scratch files with the following attributes in a SORT DEFINE:

- SCRATCH
- SCRATCHON
- NOSCRATCHON

FastSort uses these attributes, if they exist, to build a pool of scratch volumes by inclusion and by exclusion. For more information on how to specify values for these attributes, see Section 7, Using SORT and SUBSORT DEFINEs .

Figure 9-1 shows how FastSort uses DEFINEs to build a scratch volume pool.

Figure 9-1. How FastSort Reads Scratch Volume DEFINEs



FastSort first reads the SCRATCH attribute for the name of an initial scratch file or scratch volume. If no SCRATCH file or volume is specified or if the file or volume becomes full, FastSort reads the SCRATCHON attribute for acceptable overflow scratch volumes. FastSort supports up to 32 total scratch volumes: one initial volume in the SCRATCH attribute and up to 31 SCRATCHON overflow volumes.

Note. FastSort uses up to 100 percent of the disk space on volumes you specify in the SCRATCH and SCRATCHON attributes. Therefore, if you explicitly specify scratch volumes, ensure that other processes do not currently require disk space on those volumes.

If the scratch file or volume specified in SCRATCH becomes full and no SCRATCHON volumes are specified, FastSort reads the NOSCRATCHON attribute for volumes that should not be used for overflow scratch files. You can specify up to 32 NOSCRATCHON volumes.

The SCRATCHON and NOSCRATCHON attributes are mutually exclusive. If you specify SCRATCHON, you cannot exclude volumes from the pool with NOSCRATCHON. Likewise, if you specify NOSCRATCHON, you cannot specify volumes for the pool with SCRATCHON.

How Volume Characteristics Affect Selection

After checking DEFINEs for scratch volume information, FastSort creates scratch files on volumes on your system with the following features:

- The primary disk process running in the CPU where SORTPROG is running
- The fewest number of currently open scratch files
- The greatest amount of free disk space

FastSort automatically excludes $SYSTEM and volumes that:

- Contain less than 1 MB of free disk space
- Contain TMF audit trail files
- Are protected by the Safeguard product

After FastSort chooses a scratch volume, it continues to create additional scratch files on that volume until the volume is 80 percent full. When the scratch volume becomes 80 percent full, FastSort creates the next scratch file on a new volume from the pool.

Table 9-1. How FastSort Chooses Scratch Volumes

What You Specify:	How FastSort Responds:
Nothing: no scratch file, no DEFINEs with scratch attributes, no scratch volumes, no restrictions on scratch volumes	Uses volume characteristics to select a scratch volume. FastSort creates scratch files on this volume until it is 80 percent full, selects another scratch volume if necessary, and so on.
A scratch file	Uses the file until it becomes full. If the user manually creates the scratch file and it becomes full, FastSort tries to increase MAXEXTENTS and continue using the file. If overflow scratch files are needed, FastSort creates them on the current volume until it is 80 percent full. Then FastSort uses volume characteristics to choose another scratch volume. FastSort creates scratch files on the new volume until it is 80 percent full, and so on.
A scratch volume	Creates scratch files on the volume until it is 100 percent full. Then FastSort uses volume characteristics to choose another scratch volume and creates scratch files on the second volume until it is 80 percent full, and so on.
A list of scratch volumes	Resolves any wild-card characters in the scratch volumes list and assigns scratch files to the volumes in a sequential fashion. If necessary, FastSort creates scratch files on these volumes until they are 100 percent full. Then FastSort uses volume characteristics to choose additional scratch volumes, if needed.
A scratch file or scratch volumes and restrictions on scratch volumes with NOSCRATCHON	Uses the scratch file or scratch volumes specified and does not use the volumes specified with NOSCRATCHON. If you specify scratch volumes, FastSort fills them up to 100 percent full.
Only restrictions on scratch volumes with NOSCRATCHON	Ignores the volumes specified in NOSCRATCHON and uses characteristics to select a scratch volume. FastSort creates scratch files on the volume until it is 80 percent full. Then FastSort uses volume characteristics to choose another scratch volume, creates scratch files on the second volume until it is 80 percent full, and so on.
A scratch file or volume, or a scratch file and scratch volumes, restrictions on scratch volumes, a CREATE INDEX configuration file, and DEFINEs	Uses values in the configuration file. The values and options specified in the configuration file override those specified in DEFINEs. FastSort fills the specified scratch volumes up to 100 percent full.

Each scratch file extent can be up to 2048 pages, or 4 KB. For scratch files that FastSort creates, the default extent size is 4 KB and MAXEXTENTS is 978 extents. Depending on extent sizes, a nonpartitioned scratch file can be up to 1 TB in size.

Selecting a Scratch Volume for Parallel Sorts

For parallel sorts, each subsort process uses its own initial and overflow scratch files. A distributor-collector process does not usually require scratch files.

Use the SCRATCH attribute of a SUBSORT DEFINE to specify an initial scratch file for each subsort process. If you specify a fully qualified file name for this attribute, you must specify a unique scratch file for each subsort process. You cannot specify a single scratch file, or different partitions of a single scratch file, for more than one subsort.

If you want FastSort to automatically manage scratch space for a parallel sort operation, specify only a volume name in the SCRATCH attribute of the distributor-collector process SORT DEFINE.

Using the =_SORT_DEFAULTS DEFINE for Parallel Sorts

Follow these guidelines if you use only a =_SORT_DEFAULTS DEFINE to configure a parallel sort operation.

Each subsort in a parallel sort operation must use a distinct scratch file. If more than one subsort process uses a single scratch file, disk space and contention problems can result. Therefore, if you use the =_SORT_DEFAULTS_DEFINE to configure a parallel sort operation, specify only a volume name for the SORT SCRATCH attribute. Do not specify a fully qualified file name for this attribute.

Specifying Overflow Scratch Volumes for Subsorts

For large parallel sorts or when data is distributed unevenly across partitions, you can specify overflow scratch volumes for subsorts. In the SORT DEFINE that configures the distributor-collector process, specify a SCRATCHON list of overflow scratch volumes. When you specify SCRATCHON volumes for the distributor-collector process, the pool of scratch volumes is automatically available for subsorts.

When you load the partitions of an index in parallel, you should specify scratch files and volumes in a configuration file. If you do not specify a scratch file in the CREATE INDEX configuration file, FastSort uses the scratch volumes specified in the =SORT_DEFAULTS DEFINE, if any.

When you use the LOAD utility to load data into a file or table, you should specify a scratch file or volume in the SCRATCH option. If you do not specify a scratch file in the LOAD SCRATCH option, FastSort uses scratch volumes specified in the =SORT_DEFAULTS DEFINE, if any.

Using a Partitioned Scratch File

A partitioned scratch file is a single scratch file partitioned across multiple disk volumes. The multiple volumes can exist on separate nodes. A partitioned scratch file functions in essentially the same manner during a sort operation as multiple scratch files. While the maximum size of a nonpartitioned scratch file is 1 TB if it is created by the user and up to 2 GB otherwise, a partitioned scratch file can be greater than 1 TB.

Partitioned scratch files are especially useful when:

- You want to allocate all scratch space before the sort operation begins
- There is not enough space on any single disk for a scratch file
- The existing disk space is too fragmented to hold a default scratch file extent
- The sort operation requires an initial scratch file that does not fit on one volume

To use a partitioned scratch file, you first use the FUP CREATE command to manually partition and create the file. The syntax for creating a partitioned scratch file at a TACL prompt is:

```
FUP CREATE filename, PART (partition-num , [\node.]$volume
          [,pri-extent-size [, [sec-extent-size ]]]),...
```

filename

is the name of the file to create. If you specify a partial file name, the TACL command interpreter uses the current node, volume, and subvolume.

PART

sets options for each partition. Enclose options for each partition with parentheses and separate them with commas.

partition-num, [\node.]$*volume*

identifies the partition and specifies a location. Specify an integer from 1 to 15 for *partition-num*. Specify a *volume* for the partition location. You can also specify a *node*. However, for optimal performance, locate scratch files on the node where SORTPROG is running.

pri-extent-size, *sec-extent-size*

specifies the primary and secondary extent sizes for a partition. The default primary extent size is one page, or 2048 bytes. If you specify no secondary extent size or zero extents, *sec-extent size* defaults to the size of the primary extent. The value you specify can be in pages, bytes, or megabytes (MB). The default extent unit is pages. The maximum value is 65,535 pages, or 134 MB.

The following syntax creates the file SCRATCH with two secondary partitions:

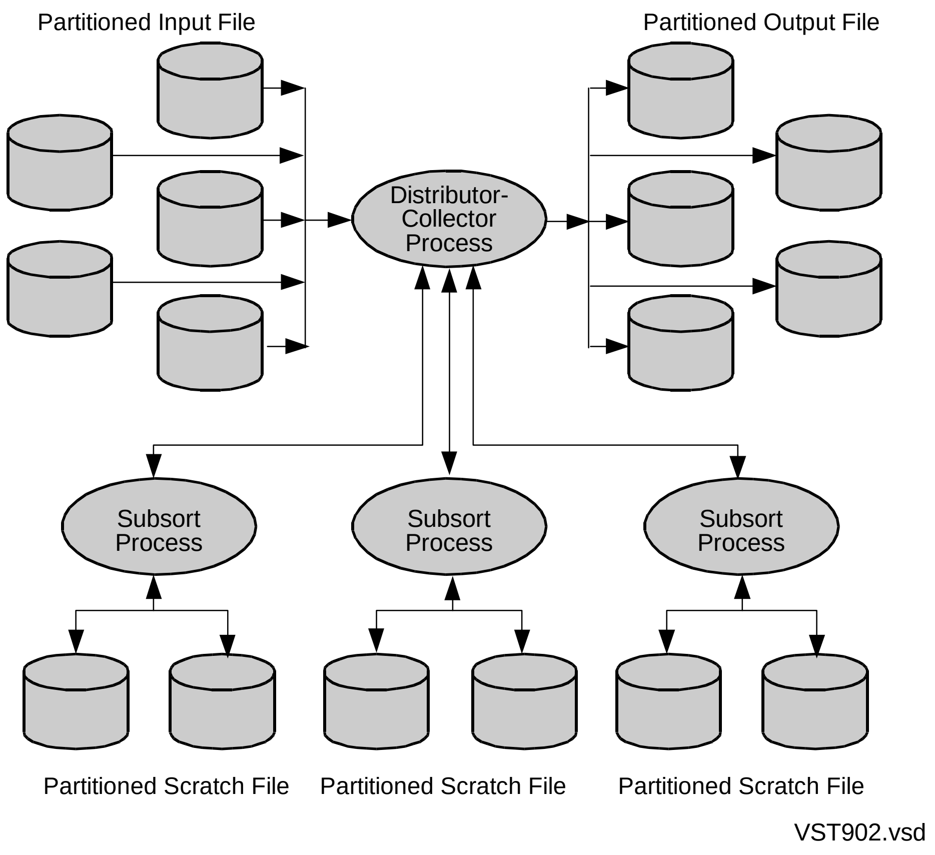
```
FUP CREATE SCRATCH, PART (1, $data3, 64, 8),
                    PART (2, $data4, 64, 8)
```

In this example, a primary file partition, SCRATCH, is created on the current node, volume, and subvolume. Two secondary partitions, also named SCRATCH, are created on $data3.<`current-subvol-name`> and $data4.<`current-subvol-name`> on the current node.

You size a partitioned scratch file in the same manner as a non-partitioned scratch file. To calculate the size of each scratch file partition, use the formula in Manually Creating a Scratch File on page 9-2. Note that the file must be unstructured. For more information about the CREATE command, see *File Utility Program (FUP) Reference Manual*.

After you partition and create the scratch file, use one of the methods listed at the beginning of this section to specify the file to FastSort. You can use partitioned scratch files for both serial and parallel sort operations. Figure 9-2 shows a parallel sort run with a 1 GB input file, three subsort processes, three partitioned scratch files and a partitioned output file.

Figure 9-2. Partitioned Scratch Files in Parallel Sorting



Using Swap Files

A swap file is the disk file used for data swapping during a sort or merge run. Data swapping is the process of copying data between physical memory and storage.

Swapping, or paging, occurs when the extended memory segment is larger than the available physical memory. Swapping also occurs when processes contend for available memory. To minimize swapping, specify less extended memory in one of the following:

- The MINSPACE, MINTIME, or SEGMENT parameter of the RUN command
- The *`flags`* parameter of the SORTMERGESTART procedure
- The SEGMENT attribute of a SORT or SUBSORT DEFINE

You can also move the sort process to a processor with a lighter load or more physical memory available.

Locating the Swap File

The swap file for FastSort is always on the local node. The default swap file location is the current scratch volume, if the scratch file is local. For remote scratch files, the default swap volume is the volume where the program file is running.

However, for optimal performance it is best to locate the swap file on a less busy volume. You can specify another location for a swap file in:

- The SWAP parameter of the RUN command
- The *`process-start`* parameter of the SORTMERGESTART procedure
- The SEGMENT attribute of a SORT or SUBSORT DEFINE
- The SWAP option in a parallel CREATE INDEX configuration file

Specifying a Swap File for Parallel Sorts

Each subsort in a parallel sort operation must use a distinct swap file. If more than one subsort process uses a single swap file, disk space and contention problems can result.

For example, if you specify a fully-qualified file name for the SUBSORT SWAP attribute of a SORT DEFINE, you must specify a unique swap file for each subsort.

If you use the =_SORT_DEFAULTS_DEFINE to configure a parallel sort operation, specify only a volume name for the SORT SWAP attribute. Do not specify a fully qualified file name for this attribute.

Using VLM

The Very Large Memory (VLM) option increases the amount of extended memory FastSort can use to sort records. If VLM is on, FastSort can use up to 127.5 MB of extended memory, if available. FastSort uses the additional extended memory either to complete the sort in a single pass or to store partial information until the sort is complete.

Without VLM, the maximum number of records that FastSort can sort in memory is 32,767. This limit applies regardless of the amount of memory available. With VLM, available memory and extended segment size determine the number of records that can be sorted in memory.

Turning On VLM

Depending on your system configuration, memory usage, and the interface to FastSort you use, the VLM option can help improve sort performance. VLM is off by default because it can use more physical memory and does not always improve performance. You can turn on VLM from:

- A SORT DEFINE, including the =_SORT_DEFAULTS DEFINE
- The SORTMERGESTART procedure

For information on which method takes precedence, see Determining the Precedence of DEFINEs on page 7-1.

When VLM is on, the maximum amount of extended memory for sorting is 127.5 MB, or 62,255 pages. This memory limit overrides any value you otherwise specify for segment size. When VLM is off, the maximum extended memory FastSort can use is 67 MB, or 32,767 pages.

Do not use VLM for parallel sort or load operations.

How VLM Affects Swap Files

A larger extended memory segment requires a larger swap file. Using VLM can cause an increase in data swapping if SORTPROG competes with other processes for memory. If increased swapping impacts performance, use one of these strategies:

- Use the SEGMENT attribute or parameter to specify a smaller extended segment
- Use the CPU attribute to specify a less-busy CPU
- Use the SWAP attribute to move the swap file to a different disk volume
- Turn VLM off to use the default extended memory and disk space utilization

For more information about FastSort swap files, see Using Swap Files on page 9-10.

How VLM Affects Scratch Files

VLM can help reduce the disk space FastSort uses for scratch files. With a larger extended memory segment, FastSort can perform some sort operations entirely in memory. Sorts performed in memory do not require scratch files.

VLM can also increase performance for sorts that do require scratch files. For sorts that require an intermediate merge pass, FastSort uses the additional memory to store partial information. The additional storage space reduces reads and writes to scratch files.

How VLM Affects Statistics

When VLM is on, the FastSort statistics format changes slightly.

For interactive FastSort, BUFFER PAGES changes from an INT to an INT(32) value when VLM is on. In this case, BUFFER PAGES can have a value greater than 32,767, the maximum extended memory segment you can manually specify. BUFFER PAGES can also be -1 as a result of VLM.

For programmatic FastSort, a parameter in the SORTMERGESTATISTICS array tells FastSort to return the larger statistics format when VLM is on. If the parameter `flag1` is present and set to 1, FastSort converts BUFFER PAGES to an INT(32) value before placing it in the statistics array. If you specify a value other than 0 or 1 for `flag1`, FastSort returns error 150 (INVALID STATISTICS FLAG VALUE SPECIFIED). For applications that use VLM, set `flag1` to 1 to get accurate statistics when BUFFER PAGES is greater than 32,767.

For more information about SORTMERGESTATISTICS, see Section 5, Using FastSort System Procedures.

Calculating Data Stack Space

If you invoke FastSort from an application program, sort complexity determines the amount of data stack space FastSort requires. Follow the guidelines in the following table to calculate data stack space requirements.

Operation	Description	Additional Space
Simple	Less than 5 keys, no subsorts, 1 input file	2 pages
Medium	Greater than 5 keys, either subsorts or multiple input files	3 pages
Complex	Greater than 5 keys, subsorts, multiple input files	4 pages

To allocate this additional space in an application, use one of the following methods:

- For a TAL application, use the DATAPAGES compiler directive during compilation. Specify DATAPAGES 64 to allocate the maximum amount.
- For all applications, use the Binder SET EXTENDSTACK command after compilation. Specify 64 PAGES to allocate the maximum amount.
- When you run the program, specify 64 pages for the MEM option of the RUN command. If you run the program from another application, specify 64 for the PROCESS_CREATE_ or NEWPROCESS[NOWAIT] `memory-pages` parameter.
- Move user data from the user data segment to an extended data segment to free up more data stack space for the call to SORTMERGESTART.

For information about TAL compiler directives, see the *TAL Reference Manual*. For information about the Binder SET command, see the *Binder Manual*.

Other Data Stack Space Considerations

In addition to the requirements listed above, if you specify either the SCRATCHON or NOSCRATCHON attributes in a SORT DEFINE, FastSort requires up to 138 additional words (276 bytes) of stack space. To learn how FastSort uses this space to build a pool of scratch volumes, see Table 5-1 on page 5-5.

If your application process starts a new process, FastSort also requires 30 to 35 additional words of stack space to support the PROCESS_CREATE_ procedure.

Managing Sort Failures

If a sort operation fails, the cause of the failure is usually stated in the error message FastSort returns. Most sort failures are caused by insufficient workspace. For more information on how to set up scratch and swap files, see Managing Sort Workspace on page 9-1. This subsection recommends strategies for managing failures that are not caused by insufficient sort workspace.

Verifying Version Compatibility

A sort operation might fail if you run versions of FastSort and other NonStop software that are incompatible. Incompatible versions are likely cause of failure, for example, if SORTPROG runs on one node, and a software component that affects FastSort runs on a second node.

For SQL/MP sort operations, you might receive error 121 (INCOMPATIBLE SQL VERSION) when the sort fails due to incompatible versions. In other cases, the error message you receive might not directly refer to a version problem.

If you suspect a version problem, check versions of the operating system, SORTPROG, SORT, SQL/MP, and other NonStop software products for compatibility. Run the VPROC utility to determine software versions. The syntax for the VPROC utility is:

```
VPROC object-file
```

In VPROC syntax, *`object-file`* is the volume, subvolume, and file name of the program object file. For example, to determine the version of SORTPROG on your local node, type:

```
VPROC $SYSTEM.SYSnn.SORTPROG
```

at a TACL prompt. The operating system returns version information in the following format:

```
VPROC-T9617D30-(31 OCT 94) SYSTEM \TSII Date 17 JUL 1995,
14:54:38
COPYRIGHT TANDEM COMPUTERS INCORPORATED 1991, 1987, 1989
```

```
$SYSTEM.SYS01.SORTPROG
     Binder timestamp:   10NOV94 10:12:17
    Version procedure:   T9620D30^31OCT94^AAU^31OCT94
           Target CPU:   TNS, TNS/R
```

Compare the information VPROC returns with version information in the softdocs shipped with your NonStop software. These documents contain product and RVU numbers for the specific product they describe. Softdocs also contain software compatibility information. If you do not have access to these documents, contact your system manager.

To determine your operating system version, check the $SYSTEM.SYS*nn*. CONFLIST file. This file is generated by SYSGEN when you install a version of the NonStop Kernel. The first page of the file contains operating system RVU and compatibility information. If you do not have read access to this file, contact your system manager.

To determine the version of SQL/MP on your node, execute the following command from within SQLCI:

```
GET VERSION OF SYSTEM;
```

For more information about the GET VERSION statement, see *SQL/MP Reference Manual.*

Saving Failure Information

If a sort fails and you cannot quickly identify the cause, you should save information about the sort process. This information will help diagnose the reason for failure if you have to contact your service provider.

You can automatically capture information about any failed process in a disk file. If a sort process terminates abnormally, this file contains valuable information about conditions at the point of termination. To automatically create a save file, you use the Binder utility to turn SAVEABEND ON.

To start the Binder utility, type

```
BIND
```

at a TACL prompt. The syntax for creating a SAVEABEND file for a SORTPROG or RECGEN process at the Binder prompt is:

```
CHANGE SAVEABEND ON IN [ SORTPROG | RECGEN ]
```

To turn on SAVEABEND, you must have permission to write to the object file. For FastSort, the object file is SORTPROG or RECGEN. If you do not have write permission to these files, contact your system manager.

You must direct the operating system to create a save file before a failure occurs. If SAVEABEND is not already set to ON on your local node when a sort fails, you must first turn on SAVEABEND and then duplicate the failure in order to save information in a save file.

Note. Turning SAVEABEND ON also sets the Binder INSPECT option to ON. For more information about the Binder CHANGE command, see *Binder Manual*.

To exit the Binder utility, type

```
EXIT
```

The save file contains information about the process environment at the time of termination, including:

- Names of all open files
- A copy of the data space at the time the process terminated
- Name of the process and a timestamp for the time of termination

The default save file location is the location of the specified object file. The save file name is always of the format ZZSA*nnnn*.

You can also use the Inspect SAVE and PR commands to save the environment of a failed process. Like the Binder utility, you must direct Inspect to save failure information before a failure occurs. For more information about these commands, see *Inspect Manual*.

Automating FastSort Tasks

One way to automate FastSort tasks is to use a command file. A command file, also sometimes called an OBEY file, is an EDIT file that contains a series of commands. When you execute the file, commands in the file are automatically executed. Use a command file to automate tasks that:

- Are repetitive
- Require many commands and few decisions
- Can cause serious problems if not properly executed

For example, you might regularly perform these tasks:

- Load data from one SQL/MP table into another
- Execute a query that causes a sort-merge join of a large SQL/MP table

The load and query operations each require an SQL statement and a =_SORT_DEFAULTS DEFINE. To reduce execution time, you can specify the commands required for each task in an EDIT file. Then execute the file when you need to perform the task. To automate configuration, you could set up a separate =_SORT_DEFAULTS DEFINE for each task. Then either enable the appropriate DEFINE in the command file or specify DEFINE commands directly in the file.

Automating DEFINEs

The following examples show how you can use command files to set up FastSort DEFINEs.

Using a Command File to Set DEFINEs from TACL

The following is an example of a TACL command file that sets SORT DEFINEs for an interactive sort operation:

```
DELETE DEFINE =SORT_ONE
SET DEFMODE ON
SET DEFINE CLASS SORT
SET DEFINE SCRATCH $data.fastsort.scratch
SET DEFINE BLOCK 28762
SET DEFINE SCRATCHON ($data??)
SET DEFINE SEGMENT 256
SET DEFINE SWAP $data.fastsort.swapfile
SET DEFINE PRI 80
ADD DEFINE =SORT_ONE
```

Note that the first line of the command file deletes the =SORT_ONE DEFINE, if it already exists. This step is optional and ensures that only the values you specify for =SORT_ONE in the command file affect this sort operation. The SET DEFMODE ON command enables DEFINEs for the current TACL session.

To execute this command file, type OBEY *filename* at a TACL prompt.

Using a Command File to Set DEFINEs from SQLCI

You can use the OBEY command to execute a command file for SQL/MP sorts from your SQLCI prompt. The following is an example of a command file that sets up a =_SORT_DEFAULTS DEFINE from within SQLCI:

```
DELETE DEFINE =_SORT_DEFAULTS;
SET DEFMODE ON;
SET DEFINE CLASS SORT;
OBEY SCRATCH1;
SET DEFINE BLOCK 57524;
SET DEFINE CPU 8;
SET DEFINE MODE MINTIME;
SET DEFINE SCRATCHON ($data2, $data4);
SET DEFINE SWAP $spare;
ADD DEFINE =_SORT_DEFAULTS;
```

Nesting Command Files in SQLCI

A command file that you execute from within SQLCI can execute another command file. You can nest up to four command files in this manner to simplify configuration changes. For example, the command file SCRATCH1 named in the previous file configures the =_SORT_DEFAULTS DEFINE for a parallel sort operation. SCRATCH1 specifies only volume names for scratch and swap files, as follows:

```
SET DEFINE SCRATCH $DATA2
SET DEFINE SWAP $SPARE2
```

You could use the SCRATCH1 configuration if the SQL optimizer chooses a parallel plan for a query that invokes FastSort.

A second command file, SCRATCH2, configures scratch and swap space for loading data from a large SQL/MP table. It directs FastSort to use a 3 GB partitioned scratch file for the load operation, as follows:

```
SET DEFINE SCRATCH $SPARE2.SCRATCH.PART
SET DEFINE SWAP $SPARE1
```

To shift from the first configuration to the second, change the nested file name in the top-level command file.

Using the SAVE Command

To preserve the DEFINE attributes of your current SQLCI session, use the SAVE command before you exit SQLCI. This command automatically preserves SQLCI session attributes as commands in a file. For more information about the SAVE command including syntax, see *SQL/MP Reference Manual*.

A FastSort Syntax Summary

This appendix contains a syntax summary of the FastSort interactive commands and system procedures.

Interactive Commands

The FastSort interactive commands are:

```
ASC[ENDING] field [ type ] [ , field [ type ] ]...
```

```
CLEAR { ALL                 }
      { ASC[ENDING]         }
      { COLLATE             }
      { CPUS                }
      { DESC[ENDING]        }
      { FROM [ filename ]   }
      { KEYS                }
      { NOTCPUS             }
      { SUBSORT             }
      { TO                  }
```

```
COLLATE filename
```

```
COLLATEOUT filename
CPUS [ ALL      ]
     [ cpu-list ]
```

```
DESC[ENDING] field [ type ] [ , field [ type ] ]...
```

```
EXIT
```

```
FC
```

```
FROM [ in-file ] [ , EXCL[USION] mode ]...
                 [ , FILE count       ]
                 [ , MERGE            ]
                 [ , RECORD length    ]
```

```
HELP [ ASC[ENDING]  ]
     [ CLEAR        ]
     [ COLLATE      ]
     [ COLLATEOUT   ]
     [ CPUS         ]
     [ DESC[ENDING] ]
     [ FROM         ]
     [ HELP         ]
     [ NOTCPUS      ]
     [ RUN          ]
     [ SAVE         ]
     [ SHOW         ]
     [ SUBSORT      ]
     [ TO           ]
```

```
NOTCPUS cpu-list

RUN                  [scratch-file |scratch-vol]
                     [ , AUTOMATIC              ]
                     [ , BLOCK size             ]
                     [ , CPU processor          ]
                     [ , MEM memory             ]
                     [ , MINSPACE               ]
                     [ , MINTIME                ]
                     [ , PRI priority           ]
                     [ , { REMOVEDUPS | REMD } ]
                     [ , DEFINE define-name     ]
                     [ , SEGMENT size           ]
                     [ , PROGRAM file           ]
                     [ , SWAP file              ]
                     [ , NOSCRATCHON(scratch-vol,scratch-
vol,...)]
                     [ , SCRATCHON(scratch-vol [,scratch-
vol]...)]

SAVE { ALL                }
     { ASC[ENDING]        }
     { COLLATE            }
     { CPUS               }
     { DESC[ENDING]       }
     { FROM [ filename ] }
     { KEYS               }
     { NOTCPUS            }
     { SUBSORT            }
     { TO                 }

SHOW { ALL                }
     { ASC[ENDING]        }
     { CPUS               }
     { DESC[ENDING]       }
     { FROM [ filename ] }
     { KEYS               }
     { NOTCPUS            }
     { SUBSORT            }
     { TO                 }

SUBSORT scratch-file [ , BLOCK size    ]...
                     [ , CPU processor ]
                     [ , MEM memory    ]
                     [ , PRI priority  ]
                     [ , SEGMENT size  ]
                     [ , PROGRAM file  ]
                     [ , SWAP file     ]

TO [ out-file ] [ , EXCL[USION] mode  ]...
                [ , KEYS               ]
                [ , PERMUTATION        ]
                [ , TYPE file-type     ]
                [ , NOPURGE            ]
                [ , SLACK percentage   ]
                [ , DSLACK percentage  ]
                [ , ISLACK percentage  ]
```

FastSort Procedures

The FastSort system procedures are:

```
{ status := } SORTBUILDPARM ( ctlblock                 ! i
{ CALL      }                ,[cpu-mask]               ! i
                             ,[not-cpu-mask]           ! i
                             ,[buffer]                 ! i
                             ,[buffer2]                ! i
                             ,[buffer-length]          ! i
                             ,[build-flags]            ! i
                             ,[define-name]            ! i
                               !reserved1!
                               !reserved2!
                             ,[scratchvols ] )         ! i

{ length := } SORTERROR     ( ctlblock                 ! i
{ CALL      }                ,buffer  )                ! o

{ status := } SORTERRORDETAIL( ctlblock )              ! i
{ CALL      }

{ length := } SORTERRORSUM  ( ctlblock                 ! i
{ CALL      }               ,[ buffer ]                ! o
                            ,[ error-code ]            ! o
                            ,[ error-source ]          ! o
                            ,[ subsort-index ]         ! o
                            ,[ subsort-id ] )          ! o

{ status := } SORTMERGEFINISH ( ctlblock               ! i
{ CALL      }                 ,[ abort ]               ! i
                              ,[ spare1 ]              !
reserved
                              ,[ spare2 ] )            !

reserved

{ status := } SORTMERGERECEIVE ( ctlblock              ! i
{ CALL      }                 ,[ record-loc ]          ! o
                              ,   length               ! o
                              ,[ spare1 ]              !
reserved
                              ,[ spare2 ]              !
reserved
                              ,[ record-loc-ext ]      ! o

{ status := } SORTMERGESEND  ( ctlblock                ! i
{ CALL      }                 ,[ record-loc ]          ! i
                              ,   length               ! i
                              ,[ stream-id ]           ! o
                              ,[ spare1 ]              !
reserved
                              ,[ spare2 ] )            !
reserved
                              ,[ record-loc-ext ]      ! i
```

```
{ status := } SORTMERGESTART ( ctlblock                  ! i
{ CALL      }              ,  key-block                 ! i
                           ,[ num-merge-files ]         ! i
                           ,[ num-sort-files ]          ! i
                           ,[ in-file-name ]            ! i
                           ,[ in-file-exclusion-mode ]  ! i
                           ,[ in-file-count ]           ! i
                           ,[ in-file-record-length ]   ! i
                           ,[ format ]                  ! i
                           ,[ out-file-name ]           ! i
                           ,[ out-file-exclusion-mode ] ! i
                           ,[ out-file-type ]           ! i
                           ,[ flags ]                   ! i
                           ,[ errnum ]                  ! o
                           ,[ errproc ]                 ! i
                           ,[ scratch-file-name ]       ! i
                           ,[ scratch-block ]           ! i
                           ,[ process-start ]           ! i
                           ,[ max-record-length ]       ! o
                           ,[ collate-sequence-table ]  ! i
                           ,[ dslack ]                  ! i
                           ,[ islack ]                  ! i
                           ,[ flags2 ]                  ! i
                           ,[ subsort-count ]           ! i
                           ,[ spare5 ]  )               !
reserved

{ status := } SORTMERGESTATISTICS ( ctlblock            ! i
{ CALL      }                      ,length              ! i, o
                                   ,statistics          ! o
                                   ,[flag1 ]            ! i
                                   ,[spare1 ]           !
reserved
```

B FastSort Error Messages

This appendix lists the FastSort error messages in three lists:

- An alphabetic list of programmatic messages starting on B-1
- A numeric list of programmatic messages starting on B-6
- An alphabetic list of interactive messages (interactive messages are not numbered) starting on B-35

The numeric list of programmatic error messages and the alphabetic list of interactive error messages include the text for the error code, a possible cause, and recovery strategies. This appendix also includes effect information for error messages that can occur when users invoke FastSort transparently from SQL/MP.

To determine appropriate recovery action for some of these errors, see *Guardian Procedure Errors and Messages Manual*, which has information about the file-system and NEWPROCESS error codes that accompany FastSort error codes.

Whenever an error occurs, the SORTPROG process stops. After you take recovery action, you need to start the process again.

You can specify a TAL procedure for FastSort to call when an error occurs. For more information about creating and specifying a procedure for error recovery, see Writing a User Error Procedure on page 5-37.

Alphabetic List of Programmatic Messages

Listed below are the programmatic FastSort error messages in alphabetic order including the corresponding FastSort error code for each message. These messages are listed numerically by FastSort error code later in this appendix.

Error Code	**Message Text** (page 1 of 6)
33	A CONTROL OPERATION HAS FAILED.
26	A KEY FIELD LOCATION EXCEEDS THE RECORD SIZE.
36	A POSITION HAS FAILED IN A SCRATCH FILE.
32	A READ HAS FAILED FROM A SCRATCH FILE.
31	A READ HAS FAILED FROM THE FROM FILE.
28	A SCRATCH FILE CANNOT BE OPENED.
48	A SIGNED ASCII NUMERIC KEY IS LARGER THAN 32 BYTES.
24	A TEMPORARY TO FILE IS TOO SMALL.
53	A TO FILE MAY NOT BE A FILE TO BE MERGED.
30	A WRITE HAS FAILED TO A SCRATCH FILE.
29	A WRITE HAS FAILED TO THE TO FILE.
34	AN EDITREAD HAS FAILED FROM THE FROM FILE.

Error Code	**Message Text** (page 2 of 6)
4	AN ERROR HAS PREVENTED CREATION OF THE SORT PROCESS.
39	AN INPUT RECORD EXCEEDED THE RECORD SIZE.
59	AN INPUT RECORD IS TOO SMALL.
81	BLOCKED INTERFACE NOT ALLOWED WITH MERGE.
133	CANNOT INCREASE THE SCRATCH FILE SIZE.
57	COLLATING SEQUENCE TABLE MUST BE PRESENT.
20	COMMUNICATIONS WITH SORTPROG HAVE BROKEN DOWN.
21	COMMUNICATIONS WITH SORTPROG WERE GARBLED.
78	COMMUNICATIONS WITH SUBSORT PROCESS HAVE FAILED.
5	COMMUNICATIONS WITH THE SORT PROCESS HAVE FAILED.
35	CREATION OF A SCRATCH FILE HAS FAILED.
37	CREATION OF THE TO FILE HAS FAILED.
123	DATETIME CONVERSION FIELD NOT FOUND.
99	DEFAULT DEFINE IS NOT OF CLASS SORT.
105	DEFINE HAS BEEN SPECIFIED BUT DEFMODE IS OFF.
50	EDIT FILES MAY NOT BE TO FILES.
122	ERROR DETERMINING SQL VERSION.
124	ERROR FROM DATETIME CONVERSION FIELDS.
112	ERROR FROM SQL FILESYSTEM VALIDATION ROUTINES.
108	ERROR IN DM BLOCK FORMAT.
173	ERROR IN MOVEX.
104	ERROR OCCURRED WHILE ACCESSING A SORT DEFINE.
107	ERROR OCCURRED WHILE ACCESSING A SUBSORT DEFINE.
115	ERROR WHILE RETRIEVING FILE LABEL SMSQL.
64	EXTENDED SEGMENT CAN NOT BE ALLOCATED.
171	EXTENDED SEGMENT CANNOT BE DEALLOCATED.
15	FILES TO BE MERGED MUST BE SORTED.
121	INCOMPATIBLE SQL VERSION.
113	INPUT FILE FOR SORTMERGESQL NOT TYPE SQL.
130	INTERNAL ERROR OCCURRED.
119	INTERNAL SQL NULL ERROR.
74	INVALID BLOCK ADDRESS SPECIFIED.
75	INVALID BLOCK LENGTH SPECIFIED.
156	INVALID COLLATION ARRAY LENGTH.
44	INVALID CONTROL BLOCK, PROCEDURE CALL REJECTED.

Error Code	**Message Text** (page 3 of 6)
66	INVALID DATA SLACK SPECIFIED.
49	INVALID EXCLUSION MODE SPECIFIED.
72	INVALID EXTENDED SEGMENT SIZE.
51	INVALID FILE TYPE SPECIFIED FOR TO FILE.
12	INVALID FLAG OR COMBINATION OF FLAGS.
73	INVALID FORMAT OF THE PROCESS STRUCTURE.
89	INVALID FROM FILE RECORD SIZE.
139	INVALID FROM-FILE SPECIFIED TO RECGEN.
67	INVALID INDEX SLACK SPECIFIED.
65	INVALID KEY FOR KEY-SEQUENCED FILE.
175	INVALID MONITOR MESSAGE LENGTH.
77	INVALID NAME OF THE SUBSORT SCRATCH FILE.
68	INVALID NEW FLAG SPECIFIED.
56	INVALID NUMBER OF FILES TO BE SORTED OR MERGED.
69	INVALID NUMBER OF SUBSORT PROCESSES.
87	INVALID OBJECT SPECIFIED AS FROM FILE.
88	INVALID OBJECT SPECIFIED AS SWAP FILE.
86	INVALID OBJECT SPECIFIED AS TO FILE.
102	INVALID OR NON-EXISTENT USER-SPECIFIED DEFINE NAME.
135	INVALID RECGEN MESSAGE VERSION.
134	INVALID RECGEN STARTUP MESSAGE.
45	INVALID SCRATCH FILE BLOCK SIZE.
54	INVALID SCRATCH FILE NAME.
71	INVALID SORT EXECUTION MODE.
169	INVALID STATISTICS FLAG VALUE SPECIFIED.
168	INVALID STATISTICS LENGTH SPECIFIED.
138	INVALID TO-FILE SPECIFIED TO RECGEN.
62	KEY LENGTH MUST BE GREATER THAN ZERO.
101	LOGICAL NAMES NOT ALLOWED.
131	MISSING REQUIRED PARAMETERS TO PROCEDURE.
174	MONITOR VERSION AND MESSAGE LENGTH CONFLICT.
82	MORE THAN ONE SUBSORT SHOULD BE SPECIFIED.
114	NO FILES INPUT FOR SORTMERGESQL.
140	NON-EXISTENT RECGEN FROM-FILE SPECIFIED.
141	NON-EXISTENT RECGEN TO-FILE SPECIFIED.
60	NOT ENOUGH STACK FOR SORTMERGESTART.

Error Code	**Message Text** (page 4 of 6)
117	NULL KEY SPECIFIED FOR NON-SQL FILE.
118	NULLVAR KEY SPECIFIED FOR NON-SQL FILE.
125	NUMBER OF SORTPROG OPENERS EXCEEDED SPECIFIED LIMIT.
25	ONE OF THE KEY FIELDS IS OF AN UNDEFINED TYPE.
52	ONLY ONE FILE MAY BE SORTED VIA SORTMERGESEND.
116	ONLY ONE FILE CAN BE SORTED BY SORTMERGESQL.
79	PARAMETERS ARE MUTUALLY EXCLUSIVE.
126	PROCESS ALREADY OPEN AND SORTPROC_OPEN_ CALLED.
164	PROCESS CREATE DATA SEGMENT ERROR.
163	PROCESS CREATE EXTENDED SWAP FILE ERROR.
161	PROCESS CREATE LIBRARY FILE ERROR.
167	PROCESS CREATE LIBRARY FILE FORMAT ERROR.
160	PROCESS CREATE PROGRAM FILE ERROR.
166	PROCESS CREATE PROGRAM FILE FORMAT ERROR.
162	PROCESS CREATE SWAP FILE ERROR.
165	PROCESS CREATE SYSTEM MONITOR ERROR.
46	REAL NUMBER KEYS MUST BE WORD ALIGNED.
149	RECGEN CALCULATES A BAD MULTIPLE MESSAGE ADDRESS.
143	RECGEN ERROR READING BASE TABLE.
145	RECGEN ERROR WHILE PACKING RECORD.
146	RECGEN ERROR WHILE RETRIEVING PRIMARY KEY.
144	RECGEN FILE LABEL RETRIEVAL ERROR.
150	RECGEN GETS A BAD SEQUENCE NUMBER IN THE MULTIPLE START UP MESSAGE.
148	RECGEN SORTPROC_CLOSE ERROR.
136	RECGEN SORTPROC_OPEN_ ERROR.
147	RECGEN SORTPROC_SEND_ ERROR.
142	RECGEN UNABLE TO OPEN BASE TABLE.
9	RECORD LENGTH TO SORTMERGESEND IS TOO SMALL OR LARGE.
63	RESERVED FLAGS MAY NOT BE SET.
13	SCRATCH FILE MUST BE UNSTRUCTURED.
170	SEGMENTS ABOVE 32767 NOT ALLOWED WITH VLM OFF.
127	SEND MESSAGE ID MISMATCH.
152	SORTBUILDPARM_INT_ UPS PARAMETER IS INVALID OR MISSING.

Error Code	**Message Text** (page 5 of 6)
132	SORTMERGESUPREC CALLED UNEXPECTEDLY.
10	SORTMERGEFINISH HAS BEEN CALLED UNEXPECTEDLY.
8	SORTMERGERECEIVE HAS BEEN CALLED UNEXPECTEDLY.
7	SORTMERGESEND HAS BEEN CALLED UNEXPECTEDLY.
111	SORTMERGESQL CALLED UNEXPECTEDLY.
58	SORTMERGESTART CALLED UNEXPECTEDLY.
47	SORTMERGESTATISTICS HAS BEEN CALLED UNEXPECTEDLY.
128	SORTPROC_SEND_ CALLED UNEXPECTEDLY.
129	SORTPROC_CLOSE CALLED UNEXPECTEDLY.
83	SORTPROG AND SORT LIBRARY DO NOT AGREE.
100	SORTPROG MUST BE SQL LICENSED.
84	SORTPROG VERSION AND OS VERSION DO NOT AGREE.
176	SORTPROG VERSION TOO OLD; CANNOT SUPPORT OPTIONAL OPEN-ON-DEMAND FEATURE.
172	SORTPROG VERSION TOO OLD; CANNOT SUPPORT REQUIRED NEW FEATURE.
61	SPARE PARAMETERS MAY NOT BE PRESENT.
120	SQL BULKIO NOT VALID FOR INPUT FILE.
76	START OF SUBSORT PROCESS HAS FAILED.
106	SUBSORT DEFINE IS NOT OF CLASS SUBSORT.
1	THE 'CTLBLOCK' PARAMETER TO SORTMERGESTART IS REQUIRED.
2	THE 'KEYS' PARAMETER TO SORTMERGESTART IS REQUIRED.
11	THE FREE LIST FILE CANNOT BE OPENED.
23	THE FROM FILE COULD NOT BE OPENED.
42	THE MEM SIZE MUST BE IN THE RANGE 1 TO 64.
22	THE MEMORY SPACE FOR SORTING IS INSUFFICIENT.
3	THE NUMBER OF KEY FIELDS MUST BE 1 TO 63 INCLUSIVE.
43	THE PRIORITY MUST BE IN THE RANGE 1 TO 199.
80	THE PRODUCT IS NOT INSTALLED.
6	THE SORT PROCESS HAS STOPPED UNEXPECTEDLY.
27	THE TO FILE ALREADY EXISTS AND CANNOT BE PURGED.
38	THE TO FILE COULD NOT BE OPENED.
55	TOO MANY FROM FILES SPECIFIED.
110	UNEXPECTED RETURN FROM DM^GET PROCEDURE.
85	UNEXPECTED RESPONSE FROM SORTPROG.

Error Code	**Message Text** (page 6 of 6)
93	UNEXPECTED RETURN FROM LOADALTFILE PROCEDURE.
92	UNEXPECTED RETURN FROM LOADCLOSE PROCEDURE.
90	UNEXPECTED RETURN FROM LOADOPEN PROCEDURE.
91	UNEXPECTED RETURN FROM LOADWRITE PROCEDURE.
153	UPS NOT SUPPORTED IN THIS ENVIRONMENT.
154	UPS WORKSPACE BAD.
103	USER-SPECIFIED DEFINE IS NOT OF CLASS SORT.

Numeric List of Programmatic Messages

The numeric list of the FastSort programmatic error messages includes the text for the FastSort error code, the probable cause for the error, and the suggested recovery. To determine the recovery action for some errors, see *Guardian Procedure Errors and Messages Manual*, which has information about the file-system and NEWPROCESS error codes that accompany some FastSort error codes.

```
1    THE 'CTLBLOCK' PARAMETER TO SORTMERGESTART IS REQUIRED.
```

Cause. The call to SORTMERGESTART did not specify a control block.

Recovery. Specify the *`ctlblock`* parameter.

```
2    THE 'KEYS' PARAMETER TO SORTMERGESTART IS REQUIRED.
```

Cause. The call to SORTMERGESTART did not define any key fields.

Recovery. Specify the *`key-block`* parameter.

```
3    THE NUMBER OF KEY FIELDS MUST BE 1 TO 63 INCLUSIVE.
```

Cause. The number of key fields was incorrect in the call to SORTMERGESTART.

Recovery. Change the *`key-block`* parameter to define from 1 to 63 key fields.

```
4    AN ERROR HAS PREVENTED CREATION OF THE SORT PROCESS.
```

Cause. The SORTPROG program name was incorrect, or some condition in the system caused the error.

Recovery. If the SORTPROG program name is incorrect, change the SORT DEFINE PROGRAM attribute to specify the correct program name.

If there is a system error, follow the recovery recommendations in the *Guardian Procedure Errors and Messages Manual* for the operating system error code returned with this FastSort error code.

If you are using the programmatic interface, call the SORTERRORDETAILDETAIL or SORTERRORSUM procedure to display the error in a specialized 32-bit format. Read the format as follows:

Parameter Word	Bits 0 1 2 3 4 5 6 7	8 9 10 11 12 13 14 15
High-Order	File System Error or PROCESS_CREATE_ Error Subcode	
Low-Order	FastSort Input File Index	FastSort Error Code

Cause. The calling process and SORTPROG could not exchange messages.

```
5    COMMUNICATIONS WITH THE SORT PROCESS HAVE FAILED.
```

Recovery. Ensure that your process does not call SORTMERGESTART with nowait I/O and you call AWAITIOX -1 to wait on other files. If your process does not combine nowait I/O and AWAITIOX -1, follow recovery recommendations in the *Guardian Procedure Errors and Messages Manual* for the operating system error code returned with this FastSort error code. For SQL programs, follow recovery recommendations in the *SQL/MP Message Manual* for the SQLCI error code returned with this FastSort error code.

```
6    THE SORT PROCESS HAS STOPPED UNEXPECTEDLY.
```

Cause. Someone stopped the SORTPROG process, the SORTPROG process abended, or the processor (CPU) went down.

Recovery. Restart the operation, possibly in another processor.

```
7    SORTMERGESEND HAS BEEN CALLED UNEXPECTEDLY.
```

Cause. The calling process called SORTMERGESEND at the wrong time.

Recovery. Correct your program logic. For more information on the normal order of FastSort procedures, see the FastSort system library procedures table in Section 5, Using FastSort System Procedures.

```
8    SORTMERGERECEIVE HAS BEEN CALLED UNEXPECTEDLY.
```

Cause. The calling process called SORTMERGERECEIVE at the wrong time.

Recovery. Correct your program logic. For more information on the normal order of FastSort procedures, see the FastSort system library procedures table in Section 5, Using FastSort System Procedures.

```
9    RECORD LENGTH TO SORTMERGESEND IS TOO SMALL OR LARGE.
```

Cause. The calling process sent a record of the wrong length.

Recovery. Change the *length* parameter in the call to SORTMERGESEND. For more information on the description of *length*, see SORTMERGESEND Procedure on page 5-15.

```
10    SORTMERGEFINISH HAS BEEN CALLED UNEXPECTEDLY.
```

Cause. The calling process called SORTMERGEFINISH at the wrong time.

Recovery. Correct your program logic. For more information on the normal order of FastSort procedures, see the FastSort system library procedures table in Section 5, Using FastSort System Procedures.

```
11    THE FREE LIST FILE CANNOT BE OPENED.
```

Cause. SORTPROG could not open or create its free-list file, a second scratch file that FastSort sometimes creates for internal memory management.

Recovery. Follow recovery recommendations in the *Guardian Procedure Errors and Messages Manual* for the operating system error code returned with this FastSort error code.

```
12    INVALID FLAG OR COMBINATION OF FLAGS.
```

Cause. Some values the *flags* parameter used in the call to SORTMERGESTART are mutually exclusive.

Recovery. Change one or more values for the *flags* parameter. For more information about the *flags* bits, see Table 5-4 on page 5-32.

```
13    SCRATCH FILE MUST BE UNSTRUCTURED.
```

Cause. A scratch file named in the call to SORTMERGESTART is a structured file.

Recovery. Specify an unstructured scratch file.

```
15    FILES TO BE MERGED MUST BE SORTED.
```

Cause. The data in one or more files specified for merging was not in sorted order.

Recovery. Check your files to see which ones are not sorted, and specify sorting before merging for those files.

```
20    COMMUNICATIONS WITH SORTPROG HAVE BROKEN DOWN.
```

Cause. Some condition in the system halted communications.

Recovery. Follow recovery recommendations in the *Guardian Procedure Errors and Messages Manual* for the operating system error code returned with this FastSort error code.

```
21    COMMUNICATIONS WITH SORTPROG WERE GARBLED.
```

Cause. Some condition in the system interfered with communications.

Recovery. Follow recovery recommendations in the *Guardian Procedure Errors and Messages Manual* for the operating system error code returned with this FastSort error code.

```
22    THE MEMORY SPACE FOR SORTING IS INSUFFICIENT.
```

Cause. Not enough storage was available for SORTPROG to sort the data.

Recovery. Specify a larger memory size, if possible, or:

- Reduce the input for a single sort run.
- Use more than one sort run to sort the data, and then merge the sorted data in another run.
- Reduce the number of subsorts for a parallel sort operation.

```
23    THE FROM FILE COULD NOT BE OPENED.
```

Cause. SORTPROG could not open one of the input files.

Recovery. Follow recovery recommendations in the *Guardian Procedure Errors and Messages Manual* for the operating system error code returned with this FastSort error code.

```
24    A TEMPORARY TO FILE IS TOO SMALL.
```

Cause. A temporary scratch file was not large enough for SORTPROG to perform the sort or merge operation.

Recovery. Specify a larger scratch file in the call to SORTMERGESTART.

```
25    ONE OF THE KEY FIELDS IS OF AN UNDEFINED TYPE.
```

Cause. FastSort did not recognize a key-field type specified in the call to SORTMERGESTART.

Recovery. Change the key-field type in the *`key-block`* parameter. For more information on *`key-block`* description, see SORTMERGESTART Procedure on page 5-19.

```
26    A KEY FIELD LOCATION EXCEEDS THE RECORD SIZE.
```

Cause. A key field does not lie entirely within the record.

Recovery. Correct either the key-field offset or the record length.

```
27    THE TO FILE ALREADY EXISTS AND CANNOT BE PURGED.
```

Cause. The specified output file exists but is too small or has a wrong type. SORTPROG cannot purge the file and create a new one because of the file's security, current usage, or some other condition in the system.

Recovery. Use the NOPURGE option of the TO command or *flags*<14>.1 in SORTMERGESTART. If this strategy fails to resolve the problem, follow recovery recommendations in the *Guardian Procedure Errors and Messages Manual* for the operating system error code returned with this FastSort error code. If the problem is security or current usage, you can change the security for the file or prevent concurrent access to it.

```
28    A SCRATCH FILE CANNOT BE OPENED.
```

Cause. SORTPROG could not open a scratch file.

Recovery. Follow recovery recommendations in the *Guardian Procedure Errors and Messages Manual* for the file-system error number returned with this FastSort error code.

```
29    A WRITE HAS FAILED TO THE TO FILE.
```

Cause. SORTPROG could not write to the output file, probably because you or FastSort underestimated the number of input records.

Recovery. If FastSort underestimated the number of input records, you can specify the number of records or name an existing file large enough to hold the output records. Follow recovery recommendations in the *Guardian Procedure Errors and Messages Manual* for the operating system error code returned with this FastSort error code.

```
30    A WRITE HAS FAILED TO A SCRATCH FILE.
```

Cause. SORTPROG could not write to a scratch file, either because you underestimated the number of input records or because of a disk process or data flow problem.

Recovery. Ensure that sufficient scratch space exists for the sort. SORTPROG might require more overflow scratch space. For more informatio about scratch files, see Managing Sort Workspace on page 9-1.

Also check the recovery recommendations in the *Guardian Procedure Errors and Messages Manual* for the file-system error number returned with this FastSort error code.

```
31    A READ HAS FAILED FROM THE FROM FILE.
```

Cause. SORTPROG could not read an input file.

Recovery. Follow recovery recommendations in the *Guardian Procedure Errors and Messages Manual* for the operating system error code returned with this FastSort error code.

```
32    A READ HAS FAILED FROM A SCRATCH FILE.
```

Cause. SORTPROG could not read a scratch file.

Recovery. Follow recovery recommendations in the *Guardian Procedure Errors and Messages Manual* for the operating system error code returned with this FastSort error code.A control operation on the output file or on a scratch file failed.

```
33    A CONTROL OPERATION HAS FAILED.
```

Recovery. Follow recovery recommendations in the *Guardian Procedure Errors and Messages Manual* for the operating system error code returned with this FastSort error code.

```
34    AN EDITREAD HAS FAILED FROM THE FROM FILE.
```

Cause. FastSort could not read an EDIT input file.

Recovery. Follow recovery recommendations in the *Guardian Procedure Errors and Messages Manual* for the operating system error code returned with this FastSort error code.

```
35    CREATION OF A SCRATCH FILE HAS FAILED.
```

Cause. FastSort could not create a scratch file for a sort or subsort process.

Recovery. Follow recovery recommendations in the *Guardian Procedure Errors and Messages Manual* for the operating system error code returned with this FastSort error code.

```
36    A POSITION HAS FAILED IN A SCRATCH FILE.
```

Cause. FastSort could not position in a scratch file.

Recovery. Use the SORT or SUBSORT DEFINE SEGMENT attribute to allocate more memory for sorting. If this strategy does not resolve the problem, follow recovery recommendations in the *Guardian Procedure Errors and Messages Manual* for the operating system error code returned with this FastSort error code.

```
37    CREATION OF THE TO FILE HAS FAILED.
```

Cause. FastSort could not create the output file.

Recovery. Follow recovery recommendations in the *Guardian Procedure Errors and Messages Manual* for the operating system error code returned with this FastSort error code.

```
38    THE TO FILE COULD NOT BE OPENED.
```

Cause. FastSort could not open the output file.

Recovery. Follow recovery recommendations in the *Guardian Procedure Errors and Messages Manual* for the operating system error code returned with this FastSort error code.

```
39    AN INPUT RECORD EXCEEDED THE RECORD SIZE.
```

Cause. An input record was larger than the maximum input record length.

Recovery. Change the size of the record or specify a larger maximum record length in the call to SORTMERGESTART.

```
42    THE MEM SIZE MUST BE IN THE RANGE 1 TO 64.
```

Cause. The value of the memory parameter was incorrect in the call to SORTMERGESTART.

Recovery. Change the memory value in the `process-start` parameter to specify from 1 to 64 pages. For the description of `process-start`, see SORTMERGESTART Procedure on page 5-19.

```
43    THE PRIORITY MUST BE IN THE RANGE 1 TO 199.
```

Cause. The value of the priority parameter was incorrect in the call to SORTMERGESTART.

Recovery. Correct the priority value in the `process-start` parameter. For the description of `process-start`, see SORTMERGESTART Procedure on page 5-19.

```
44    INVALID CONTROL BLOCK, PROCEDURE CALL REJECTED.
```

Cause. The calling process corrupted the FastSort control block.

Recovery. Correct your program so that it does not overwrite the control block.

```
45    INVALID SCRATCH FILE BLOCK SIZE.
```

Cause. The scratch file block size was incorrect in the call to SORTMERGESTART.

Recovery. Correct the value in the *`scratch-block`* parameter. For description of *`scratch-block`*, see SORTMERGESTART Procedure on page 5-19.

```
46    REAL NUMBER KEYS MUST BE WORD ALIGNED.
```

Cause. The offset of a real numeric key-field inside the record is not on a word boundary.

Recovery. Correct either the offset or the record layout.

```
47    SORTMERGESTATISTICS HAS BEEN CALLED UNEXPECTEDLY.
```

Cause. The calling process called SORTMERGESTATISTICS at the wrong time.

Recovery. Correct your program logic. For more information on the normal order of FastSort procedures, see the FastSort system library procedures table in Section 5, Using FastSort System Procedures. If this strategy fails to resolve the problem, check error log files for originating FastSort error.

```
48    A SIGNED ASCII NUMERIC KEY IS LARGER THAN 32 BYTES.
```

Cause. A key field of a signed numeric type is too big.

Recovery. Change the key-field type in the syntax, or change the key-field format.

```
49    INVALID EXCLUSION MODE SPECIFIED.
```

Cause. An exclusion mode specified in the call to SORTMERGESTART was invalid.

Recovery. Change the value of the *`in-file-exclusion-mode`* or *`out-file-exclusion-mode`* parameter. For the descriptions of these parameters, see SORTMERGESTART Procedure on page 5-19.

```
50    EDIT FILES MAY NOT BE TO FILES.
```

Cause. The specified output file is an EDIT file.

Recovery. Change the format of the output file or specify another file that is not an EDIT file.

```
51    INVALID FILE TYPE SPECIFIED FOR TO FILE.
```

Cause. The output file type specified in the call to SORTMERGESTART was invalid.

Recovery. Change the value of the *`out-file-type`* parameter. For the description of *`out-file-type`*, see SORTMERGESTART Procedure on page 5-19.

```
52    ONLY ONE FILE MAY BE SORTED VIA SORTMERGESEND.
```

Cause. The call to SORTMERGESTART specified multiple sort files with blank names.

Recovery. Specify only one sort file or name the files. You might also need to correct the program logic.

```
53    A TO FILE MAY NOT BE A FILE TO BE MERGED.
```

Cause. The name of the output file is the same as the name of a merge file.

Recovery. Change the output file name or the merge file name..

```
54    INVALID SCRATCH FILE NAME.
```

Cause. The scratch volume did not exist, a scratch file name was specified incorrectly, or the node was not accessible.

Recovery. Specify an existing volume, correct the file name, or specify an accessible node.

```
55    TOO MANY FROM FILES SPECIFIED.
```

Cause. The number of input files exceeded the limit.

Recovery. Reduce the number of input files.

```
56    INVALID NUMBER OF FILES TO BE SORTED OR MERGED.
```

Cause. The call to SORTMERGESTART specified a negative number of files to be sorted or merged.

Recovery. Change the value of the *`num-sort-files`* or *`num-merge-files`* parameter to a positive number.

```
57    COLLATING SEQUENCE TABLE MUST BE PRESENT.
```

Cause. The call to SORTMERGESTART specified translation, but the alternate collating sequence table was missing.

Recovery. Set *flags*.<10:10> to 0 or provide an alternate collating sequence table. For the description of the *flags* parameter, see SORTMERGESTART Procedure on page 5-19.

```
58    SORTMERGESTART CALLED UNEXPECTEDLY.
```

Cause. The calling process called SORTMERGESTART at the wrong time.

Recovery. Correct your program logic. For more information on the normal order of FastSort procedures, see the FastSort system library procedures table in Section 5, Using FastSort System Procedures.

```
59    AN INPUT RECORD IS TOO SMALL.
```

Cause. A record from SORTMERGESEND was too small. Or, an input file might contain variable-length records.

Recovery. Correct your program logic to enlarge the record size, or change the input file to contain only fixed-length records.

```
60    NOT ENOUGH STACK FOR SORTMERGESTART.
```

Cause. Not enough stack was available to call SORTMERGESTART.

Recovery. To allocate more data stack space, use one of the following methods:

- For a TAL application, use the DATAPAGES compiler directive during compilation. Specify DATAPAGES 64 to allocate the maximum amount. For all applications, use the Binder SET EXTENDSTACK command after compilation.
- When you run the program, specify 64 for the MEM option of the RUN command. If you run the program from another application, specify 64 for the PROCESS_CREATE_ or NEWPROCESS[NOWAIT] *memory-pages* parameter.
- Move user data from the user data segment to an extended data segment to free up more data stack space for the call to SORTMERGESTART.

```
61    SPARE PARAMETERS MAY NOT BE PRESENT.
```

Cause. The call to SORTMERGESTART, SORTMERGESEND, or SORTMERGERECEIVE included one of the spare parameters.

Recovery. Remove the spare parameter.

```
62    KEY LENGTH MUST BE GREATER THAN ZERO.
```

Cause. The length of a key field specified in the call to SORTMERGESTART was not positive.

Recovery. Specify a positive length for the key field in the *key-block* parameter. For the description of *key-block*, see SORTMERGESTART Procedure on page 5-19.

```
63    RESERVED FLAGS MAY NOT BE SET.
```

Cause. The call to SORTMERGESTART or SORTBUILDPARM specified a flag value that you cannot set.

Recovery. Set all unused flag bits to 0. For SORTMERGESTART, Table 5-6 shows *flags* bits you can use. The only *flags2* bits you can use are <.4> and <.15>. For SORTBUILDPARM, the only *build-flags* bit you can use is <.15>.

```
64    EXTENDED SEGMENT CAN NOT BE ALLOCATED.
```

Cause. FastSort could not allocate an extended memory segment for a sort or subsort process.

Recovery. Follow recovery recommendations in the *Guardian Procedure Errors and Messages Manual* for the operating system error code returned with this FastSort error code.

```
65    INVALID KEY FOR KEY-SEQUENCED FILE.
```

Cause. The sort key field specified for a key-sequenced file is not the same as the file's primary key field. Another possible cause is that the data type of the sort key field is not UNSIGNED.

Recovery. Make sure the sort key field and the primary key field are the same. When the type of the output file is key-sequenced, make sure the type of the sort key field is UNSIGNED.

```
66    INVALID DATA SLACK SPECIFIED.
```

Cause. The data slack value in the call to SORTMERGESTART was outside the limits for the value.

Recovery. Change the *dslack* parameter. For the description of *dslack*, see SORTMERGESTART Procedure on page 5-19.

```
67    INVALID INDEX SLACK SPECIFIED.
```

Cause. The data slack value in the call to SORTMERGESTART was outside the limits for the value.

Recovery. Change the *dslack* parameter. For the description of *dslack*, see SORTMERGESTART Procedure on page 5-19.

```
68    INVALID NEW FLAG SPECIFIED.
```

Cause. The *flags2* value in the call to SORTMERGESTART was not valid.

Recovery. Correct the value of the *flags2* parameter. For the description of *flags2*, seeunder the SORTMERGESTART Procedure on page 5-19.

```
69    INVALID NUMBER OF SUBSORT PROCESSES.
```

Cause. The number of subsort processes specified in the call to SORTMERGESTART was outside the limits.

Recovery. Change the value of the *subsort-count* parameter to specify from 2 to 16 subsort processes. Because more than 8 subsort processes can cause run-time errors, HP recommends that you specify a maximum of 8 subsorts.

```
71    INVALID SORT EXECUTION MODE.
```

Cause. The *flags* parameter in the call to SORTMERGESTART specified both MINSPACE and MINTIME.

Recovery. Set either the MINSPACE flag or the MINTIME flag to 0 before you specify the *flags* parameter. For descriptions of these flags, see Table 5-4 on page 5-32.

```
72    INVALID EXTENDED SEGMENT SIZE.
```

Cause. The extended segment size specified in the call to SORTMERGESTART was out of limits.

Recovery. Correct the segment value in the *process-start* parameter. For the description of *process-start*, see SORTMERGESTART Procedure on page 5-19.

```
73    INVALID FORMAT OF THE PROCESS STRUCTURE.
```

Cause. The call to SORTMERGESTART specified that the NEWPROCESS structure be expanded but did not specify the structure; or the calling process requested parallel sorting but did not specify that the NEWPROCESS structure be expanded.

Recovery. Correct the values of the *process-start* parameter. For the description of *process-start*, see SORTMERGESTART Procedure on page 5-19.

```
74    INVALID BLOCK ADDRESS SPECIFIED.
```

Cause. A block buffer address in the call to SORTBUILDPARM was outside stack limits; or the call to SORTMERGESTART specified nowait I/O, but the call to SORTBUILDPARM specified only one of the two buffers.

Recovery. Correct the value of the *buffer* or *buffer2* parameter or both values; or omit the nowait parameter, *flags2*, from the call to SORTMERGESTART. For more information about buffers and nowait I/O, see the description of the SORTBUILDPARM Procedure on page 5-2.

```
75    INVALID BLOCK LENGTH SPECIFIED.
```

Cause. The block buffer length specified in the call to SORTBUILDPARM was outside the limits.

Recovery. Correct the value of the *buffer-length* parameter. For the description of *buffer-length*, see SORTBUILDPARM Procedure on page 5-2.

```
76    START OF SUBSORT PROCESS HAS FAILED.
```

Cause. The distributor-collector process could not start a subsort process in the processor (CPU) you specified or in any processor in the pool.

Recovery. If you specified a processor, try specifying a different one or letting FastSort select the processor. Follow recovery recommendations in the *Guardian Procedure Errors and Messages Manual* for the operating system error code returned with this FastSort error code.

```
77    INVALID NAME OF THE SUBSORT SCRATCH FILE.
```

Cause. The call to SORTMERGESTART did not specify a valid name for the scratch file of a subsort process.

Recovery. Specify a valid name in the *scratch-file-name* parameter. For the description of *scratch-file-name*, see SORTMERGESTART Procedure on page 5-19.

```
78    COMMUNICATIONS WITH SUBSORT PROCESS HAVE FAILED.
```

Cause. The distributor-collector process could not communicate with a subsort process.

Recovery. Follow recovery recommendations in the *Guardian Procedure Errors and Messages Manual* for the operating system error code returned with this FastSort error code.

```
79    PARAMETERS ARE MUTUALLY EXCLUSIVE.
```

Cause. The call to SORTMERGESEND or to SORTMERGERECEIVE specified both a buffer and an extended buffer.

Recovery. Omit either the *buffer* parameter or the *buffer-ext* parameter. For the descriptions of these parameters, see SORTMERGESEND Procedure on page 5-15 or SORTMERGERECEIVE Procedure on page 5-13.

```
80    THE PRODUCT IS NOT INSTALLED.
```

Cause. The license PROMS were not purchased for your system.

Recovery. Because of changes in the way unlicensed software is detected in C00 and later RVUs, you should not see this message. Contact your service provider.

```
81    BLOCKED INTERFACE NOT ALLOWED WITH MERGE.
```

Cause. The call to SORTBUILDPARM specified a buffer for record blocking and the call to SORTMERGESTART specified input streams for merging through SORTMERGESEND.

Recovery. Omit the *`buffer`* and *`buffer2`* parameters from the call to SORTBUILDPARM.

```
82    MORE THAN ONE SUBSORT SHOULD BE SPECIFIED.
```

Cause. Only one subsort process was specified for a parallel sort run.

Recovery. Specify at least two subsort processes.

```
83    SORTPROG AND SORT LIBRARY DO NOT AGREE.
```

Cause. Your system has components of both SORT and FastSort installed.

Recovery. Contact your system manager; or install FastSort again. Make sure the sort library procedures correspond to the product.

```
84    SORTPROG VERSION AND OS VERSION DO NOT AGREE.
```

Cause. Your system has incompatible versions of FastSort and the operating system installed.

Recovery. Contact your system manager or service provider to have the correct version of FastSort or the operating system installed on your system.

```
85    UNEXPECTED RESPONSE FROM SORTPROG.
```

Cause. The wrong program was used as a sort process.

Recovery. Correct the sort-program field of the *`process-start`* structure for the SORTMERGESTART procedure, or correct the SORT DEFINE used by the program.

```
86    INVALID OBJECT SPECIFIED AS TO FILE.
```

Cause. The output file is an SQL object.

Recovery. The TO file cannot be an SQL object. Specify a file other then an SQL object for the TO file.

```
87    INVALID OBJECT SPECIFIED AS FROM FILE.
```

Cause. A FROM file is an SQL object.

Recovery. The FROM file cannot be an SQL object. Specify a file other then an SQL object for the FROM file.

```
88    INVALID OBJECT SPECIFIED AS SWAP FILE.
```

Cause. A swap file is an SQL object.

Recovery. Specify an Enscribe file as the swap file or use the default.

```
89    INVALID FROM FILE RECORD SIZE.
```

Cause. The record size specified for an input file is greater than 4080.

Recovery. This PVU of FastSort does not support record sizes greater than 4080. There is no recovery.

```
90    UNEXPECTED RETURN FROM LOADOPEN PROCEDURE.
```

Cause. A programming error occurred on a call to an internal procedure.

Recovery. Report the internal error number returned with this FastSort error code to your service provider. The high-order word of the *error-code* parameter returned by SORTERRORSUM and SORTERRORDETAIL contains the internal error number.

```
91    UNEXPECTED RETURN FROM LOADWRITE PROCEDURE.
```

Cause. A programming error occurred on a call to an internal procedure.

Recovery. Report the internal error number returned with this FastSort error code to your service provider. The high-order word of the *error-code* parameter returned by SORTERRORSUM and SORTERRORDETAIL contains the internal error number.

```
92    UNEXPECTED RETURN FROM LOADCLOSE PROCEDURE.
```

Cause. A programming error occurred on a call to an internal procedure.

Recovery. Report the internal error number returned with this FastSort error code to your service provider. The high-order word of the *error-code* parameter returned by SORTERRORSUM and SORTERRORDETAIL contains the internal error number.

```
93    UNEXPECTED RETURN FROM LOADALTFILE PROCEDURE.
```

Cause. A programming error occurred on a call to an internal procedure.

Recovery. Report the internal error number returned with this FastSort error code to your service provider. The high-order word of the *`error-code`* parameter returned by SORTERRORSUM and SORTERRORDETAIL contains the internal error number.

```
99    DEFAULT DEFINE IS NOT OF CLASS SORT.
```

Cause. A DEFINE with the reserved name "_SORT_DEFAULTS" was created, but is not of class SORT.

Recovery. Delete the DEFINE and optionally recreate it as a SORT DEFINE.

```
100    SORTPROG MUST BE SQL LICENSED.
```

Cause. SORTPROG has not been SQL licensed.

Recovery. FUP LICENSE SORTPROG. You must have the super ID (user ID 255,255) to license a program. For more information on the FUP LICENSE command, see *File Utility Program (FUP) Reference Manual*.

```
101    LOGICAL NAMES NOT ALLOWED.
```

Cause. You used a logical DEFINE name for an input file, output file, or the scratch file.

Recovery. Use the actual file names for an input, output, or scratch file. Do not use a DEFINE name.

```
102  INVALID OR NON-EXISTENT USER-SPECIFIED DEFINE NAME.
```

Cause. The DEFINE name you specified was not valid.

Recovery. Specify a valid DEFINE name.

```
103    USER-SPECIFIED DEFINE IS NOT OF CLASS SORT.
```

Cause. The DEFINE CLASS must be SORT.

Recovery. Specify CLASS SORT in your SORT DEFINEs.

```
104    ERROR OCCURRED WHILE ACCESSING A SORT DEFINE.
```

Cause. Unacceptable DEFINE name encountered, attribute missing, or error from procedure call.

Recovery. Check that a valid DEFINE name was specified.

```
105    DEFINE HAS BEEN SPECIFIED BUT DEFMODE IS OFF.
```

Cause. DEFMODE must be on to activate DEFINEs.

Recovery. Set DEFMODE to ON or determine why DEFMODE is not ON.

```
106    SUBSORT DEFINE IS NOT OF CLASS SUBSORT.
```

Cause. The DEFINE class must be SUBSORT.

Recovery. Specify CLASS SUBSORT in your SUBSORT DEFINEs.

```
107    ERROR OCCURRED WHILE ACCESSING A SUBSORT DEFINE.
```

Cause. Unacceptable DEFINE name encountered, attribute missing, or error from procedure call.

Recovery. Check that a valid DEFINE name was specified.

```
108   INVALID DM BLOCK FORMAT FOR SORTMERGESQL.
```

Cause. A programming error occurred on a call to an internal procedure.

Effect. The SQL DDL or DML operation in progress terminates abnormally.

Recovery. Generate and save a copy of your SAVEABEND file. Then contact your service provider.

```
109    UNEXPECTED RETURN FROM SQL DM^START PROCEDURE.
```

Cause. A programming error occurred on a call to an internal procedure.

Effect. The SQL DDL or DML operation in progress terminates abnormally.

Recovery. Generate and save a copy of your SAVEABEND file. Then contact your service provider.

```
110    UNEXPECTED RETURN FROM SQL DM^GET PROCEDURE.
```

Cause. A programming error occurred on a call to an internal procedure.

Effect. The SQL DDL or DML operation in progress terminates abnormally.

Recovery. Generate and save a copy of your SAVEABEND file. Then contact your service provider.

```
111    SORTMERGESQL CALLED UNEXPECTEDLY.
```

Cause. A programming error occurred on a call to an internal procedure.

Effect. The SQL DDL or DML operation in progress terminates abnormally.

Recovery. Generate and save a copy of your SAVEABEND file. Then contact your service provider.

```
112    ERROR FROM SQL FILESYSTEM VALIDATION ROUTINES.
```

Cause. A programming error occurred on a call to an internal procedure.

Effect. The SQL DDL or DML operation in progress terminates abnormally.

Recovery. Generate and save a copy of your SAVEABEND file. Then contact your service provider.

```
113    INPUT FILE FOR SORTMERGESQL NOT TYPE SQL.
```

Cause. A programming error occurred on a call to an internal procedure.

Effect. The SQL DDL or DML operation in progress terminates abnormally.

Recovery. Generate and save a copy of your SAVEABEND file. Then contact your service provider.

```
114    NO FILES INPUT TO SORTMERGESQL.
```

Cause. A programming error occurred on a call to an internal procedure.

Effect. The SQL DDL or DML operation in progress terminates abnormally.

Recovery. Generate and save a copy of your SAVEABEND file. Then contact your service provider.

```
115    ERROR RETRIEVING SQL FILE LABEL.
```

Cause. A programming error occurred on a call to an internal procedure.

Effect. The SQL DDL or DML operation in progress terminates abnormally.

Recovery. Generate and save a copy of your SAVEABEND file. Then contact your service provider.

```
116    ONLY ONE FILE CAN BE SORTED VIA SORTMERGESQL.
```

Cause. A programming error occurred on a call to an internal procedure.

Effect. The SQL DDL or DML operation in progress terminates abnormally.

Recovery. Generate and save a copy of your SAVEABEND file. Then contact your service provider.

```
117    NULL KEY SPECIFIED FOR NON-SQL FILE.
```

Cause. Stated in the error message.

Effect. The SQL DDL or DML operation in progress terminates abnormally.

Recovery. Generate and save a copy of your SAVEABEND file. Then contact your service provider.

```
118    NULLVAR KEY SPECIFIED FOR NON-SQL FILE.
```

Cause. A programming error occurred on a call to an internal procedure.

Effect. The SQL DDL or DML operation in progress terminates abnormally.

Recovery. Generate and save a copy of your SAVEABEND file. Then contact your service provider.

```
119    INTERNAL SQL NULL ERROR.
```

Cause. A programming error occurred on a call to an internal procedure.

Effect. The SQL DDL or DML operation in progress terminates abnormally.

Recovery. Generate and save a copy of your SAVEABEND file. Then contact your service provider.

```
120    SQL BULKIO NOT VALID FOR SPECIFIED INPUT FILE.
```

Cause. A programming error occurred on a call to an internal procedure.

Effect. The SQL DDL or DML operation in progress terminates abnormally.

Recovery. Generate and save a copy of your SAVEABEND file. Then contact your service provider.

```
121    INCOMPATIBLE SQL VERSION.
```

Cause. A remote SORTPROG process does not support features required for the requested sort.

Effect. The SQL DDL or DML operation in progress terminates abnormally.

Recovery. Specify a local SORTPROG process. If this strategy does not resolve the problem, generate and save a copy of your SAVEABEND file. Then contact your service provider.

```
122    ERROR DETERMINING SQL VERSION.
```

Cause. A file-system error occurred on a system procedure call to determine the SQL version of a remote SORTPROG process.

Effect. The SQL DDL or DML operation in progress terminates abnormally.

Recovery. Correct the file-system error condition, or specify a local SORTPROG process. To correct the file-system error condition, follow the recovery recommendations in the *Guardian Procedure Errors and Messages Manual* for the file-system error code returned with this FastSort error code. If this strategy fails to resolve the problem, generate and save a copy of your SAVEABEND file. Then contact your service provider.

```
123    DATETIME CONVERSION FIELD NOT FOUND.
```

Cause. A date-time field was specified as needing date-time conversion, but no date-time field was found.

Effect. The SQL DDL or DML operation terminates abnormally.

Recovery. If you have written your own application, change the sort key values to valid field types. If this error is returned by SQLCI, contact your service provider.

```
124    ERROR FROM DATETIME CONVERSION FIELDS.
```

Cause. A programming error occurred on a call to an internal procedure.

Effect. The SQL DDL or DML operation terminates abnormally.

Recovery. Report the internal error number returned with this FastSort error code to your service provider. The high-word order of the *`error-code`* parameter returned by SORTERRORSUM and SORTERRORDETAIL contains the internal error number.

```
125    NUMBER OF SORTPROG OPENERS EXCEEDED SPECIFIED LIMIT.
```

Cause. The number of openers exceeded the specified limit. This error can be caused when too many RECGEN processes attempt to open the same SORTPROG process.

Effect. The SQL DDL or DML operation terminates abnormally.

Recovery. Attempt to reduce the openers or contact your service provider.

```
126    PROCESS ALREADY OPEN AND SORTPROC_OPEN CALLED.
```

Cause. A programming error occurred on a call to an internal procedure.

Effect. The SQL DDL or DML operation terminates abnormally.

Recovery. Generate and save a copy of your SAVEABEND file. Then contact your service provider.

```
127    SEND MESSAGE ID MISMATCH.
```

Cause. A programming error occurred on a call to an internal procedure.

Effect. The SQL DDL or DML operation terminates abnormally.

Recovery. Generate and save a copy of your SAVEABEND file. Then contact your service provider.

```
128    SORTPROC_SEND_ CALLED UNEXPECTEDLY.
```

Cause. A programming error occurred on a call to an internal procedure.

Effect. The SQL DDL or DML operation terminates abnormally.

Recovery. Generate and save a copy of your SAVEABEND file. Then contact your service provider.

```
129    SORTPROC_CLOSE CALLED UNEXPECTEDLY.
```

Cause. Stated in the error message.

Effect. None.

Recovery. No recovery is necessary.

```
130    INTERNAL SORT ERROR.
```

Cause. A programming error occurred on a call to an internal procedure.

Effect. The SQL DDL or DML operation terminates abnormally.

Recovery. Generate and save a copy of your SAVEABEND file. Then contact your service provider.

```
131    MISSING OR CONFLICTING PARAMETERS IN SORTLIB CALL.
```

Cause. Stated in the error message.

Effect. The SQL DDL or DML operation terminates abnormally.

Recovery. Check the parameters of the sort library call that resulted in this error.

```
132    SORTMERGEDUPREC CALLED UNEXPECTEDLY.
```

Cause. A programming error occurred on a call to an internal procedure.

Effect. The SQL DDL or DML operation terminates abnormally.

Recovery. Generate and save a copy of your SAVEABEND file. Then contact your service provider.

```
133    CANNOT INCREASE THE SCRATCH FILE SIZE.
```

Cause. The SORTPROG process failed to increase the maximum number of extents for the scratch file because one of the following errors occurred:

- An increase of the maximum number of extents for the scratch file would cause the file to exceed its maximum limit. FastSort also returns file system error 21 (ILLEGAL `count` SPECIFIED).
- A file-system error (other than number 21) occurred when SORTPROG tried to increase the number of extents for the scratch file.

Recovery. For the first error, create a partitioned scratch file large enough to hold all of the records. For the second error, follow the recovery recommendations in the *Guardian Procedure Errors and Messages Manual* for the file-system error number returned with the FastSort error code.

```
134    INVALID RECGEN STARTUP MESSAGE.
```

Cause. Stated in the error message.

Effect. The SQL DDL or DML operation in progress terminates abnormally.

Recovery. Check that the versions of RECGEN and SQLUTIL are compatible. If the base table was in use when FastSort returned this error, ensure that the base table is not in use and try to re-create the index. If this strategy fails to solve the problem, generate and save a copy of your SAVEABEND file. Then contact your service provider.

```
135    INVALID RECGEN MESSAGE VERSION.
```

Cause. Stated in the error message.

Effect. The SQL DDL or DML operation in progress terminates abnormally.

Recovery. If the base table was in use when FastSort returned this error, ensure that the base table is not in use and try again to create the index. If this strategy fails to solve the problem, generate and save a copy of your SAVEABEND file. Then contact your service provider.

```
136    RECGEN SORTPROC_OPEN_ ERROR.
```

Cause. The RECGEN process encountered an error while opening the SORTPROG process. The file-system error number is included in this message.

Effect. The SQL DDL or DML operation in progress terminates abnormally.

Recovery. Attempt to correct the file-system error condition and try again to create the index. If this strategy fails to solve the problem, generate and save a copy of your SAVEABEND file. Then contact your service provider.

```
137    RECGEN INTERNAL ERROR: KEYS OUT OF ORDER.
```

Cause. Stated in the error message.

Effect. The SQL DDL or DML operation in progress terminates abnormally.

Recovery. Generate and save a copy of your SAVEABEND file. Then contact your service provider.

```
138    INVALID TO-FILE SPECIFIED TO RECGEN.
```

Cause. Stated in the error message.

Effect. The SQL DDL or DML operation in progress terminates abnormally.

Recovery. Generate and save a copy of your SAVEABEND file. Then contact your service provider.

```
139    INVALID FROM-FILE SPECIFIED TO RECGEN.
```

Cause. Stated in the error message.

Effect. The SQL DDL or DML operation in progress terminates abnormally.

Recovery. Generate and save a copy of your SAVEABEND file. Then contact your service provider.

```
140    NON-EXISTENT RECGEN FROM-FILE SPECIFIED.
```

Cause. Stated in the error message.

Effect. The SQL DDL or DML operation in progress terminates abnormally.

Recovery. Generate and save a copy of your SAVEABEND file. Then contact your service provider.

```
141    NON-EXISTENT RECGEN TO-FILE SPECIFIED.
```

Cause. Stated in the error message.

Effect. The SQL DDL or DML operation in progress terminates abnormally.

Recovery. Generate and save a copy of your SAVEABEND file. Then contact your service provider.

```
142    RECGEN UNABLE TO OPEN BASE TABLE.
```

Cause. Stated in the error message.

Effect. The SQL DDL or DML operation in progress terminates abnormally.

Recovery. Attempt to correct the condition described by the file-system error number. If the base table was in use when FastSort returned this error, ensure that the base table is not in use and attempt the load operation again. If this strategy fails to solve the problem, generate and save a copy of your SAVEABEND file. Then contact your service provider.

```
143    RECGEN ERROR READING BASE TABLE.
```

Cause. The disk process or file system found an error in a base table record.

Effect. The SQL DDL or DML operation in progress terminates abnormally.

Recovery. Attempt to correct the condition described by the file-system error number. Then ensure that blocks and individual records in the base table contain no errors. If a record contains an error, correct the error and attempt the load operation again. If this strategy fails to solve the problem, generate and save a copy of your SAVEABEND file. Then contact your service provider.

```
144    RECGEN FILE LABEL RETRIEVAL ERROR.
```

Cause. The RECGEN process could not retrieve a file label from the disk process.

Effect. The SQL DDL or DML operation in progress terminates abnormally.

Recovery. Attempt to correct the condition described by the file-system error number. Check for disk hardware errors. If this strategy fails to solve the problem, generate and save a copy of your SAVEABEND file. Then contact your service provider.

```
145    RECGEN ERROR WHILE PACKING RECORD.
```

Cause. Stated in the error message.

Effect. The SQL DDL or DML operation in progress terminates abnormally.

Recovery. Generate and save a copy of your SAVEABEND file. Then contact your service provider.

```
146    RECGEN ERROR WHILE RETRIEVING PRIMARY KEY.
```

Cause. Stated in the error message.

Effect. The SQL DDL or DML operation in progress terminates abnormally.

Recovery. Generate and save a copy of your SAVEABEND file. Then contact your service provider.

```
147    RECGEN SORTPROC_SEND_ ERROR.
```

Cause. The SORTPROG process terminated during a table load operation.

Effect. The SQL DDL or DML operation in progress terminates abnormally.

Recovery. Generate and save a copy of your SAVEABEND file. Then contact your service provider.

```
148    RECGEN SORTPROC_CLOSE ERROR.
```

Cause. Internal error

Effect. The SQL DDL or DML operation in progress terminates abnormally.

Recovery. Generate and save a copy of your SAVEABEND file. Then contact your service provider.

```
149 RECGEN CALCULATES A BAD MULTIPLE MESSAGE ADDRESS.
```

Cause. Stated in the error message.

Effect. The SQL DDL or DML operation in progress terminates abnormally.

Recovery. Check that the versions of RECGEN and SQLUTIL are compatible. If the base table was in use when FastSort returned this error, ensure that the base table is not in use and try to re-create the index. If this strategy fails to solve the problem, generate and save a copy of your SAVEABEND file. Then contact your service provider.

```
150 RECGEN GETS A BAD SEQUENCE NUMBER IN THE MULTIPLE START
UP MESSAGE.
```

Cause. Stated in the error message.

Effect. The SQL DDL or DML operation in progress terminates abnormally.

Recovery. Check that the versions of RECGEN and SQLUTIL are compatible. If the base table was in use when FastSort returned this error, ensure that the base table is not in use and try to re-create the index. If this strategy fails to solve the problem, generate and save a copy of your SAVEABEND file. Then contact your service provider.

```
152    SORTBUILDPARAM_INT_ UPS PARAMETER IS INVALID OR
MISSING.
```

Cause. A UPS parameter to SORTBUILDPARM is either invalid or missing.

Effect. The SQL DLL or DML operation in progress terminates abnormally.

Recovery. Check the UPS parameters to SORTBUILDPARM.

```
153   UPS NOT SUPPORTED IN THIS ENVIRONMENT.
```

Cause. UPS cannot be used if subsorts are used, if the number of records to be sorted is greater than 32,768, or if the multiple openers feature is being used.

Effect. The SQL DDL or DML operation in progress terminates abnormally.

Recovery. Check to see if any of the above conditions is true. If so, correct the condition.

```
154    UPS WORKSPACE BAD.
```

Cause. The eye-catcher field in the UPS workspace is corrupted.

Effect. The SQL DDL or DML operation in progress terminates abnormally.

Recovery. Check that the UPS workspace is valid and not affected by the user program.

```
156    INVALID COLLATION ARRAY LENGTH.
```

Cause. The collation sequence table length you specified was not valid.

Effect. The SQL DDL or DML operation in progress terminates abnormally.

Recovery. Specify a valid length for the collation sequence table.

```
160    PROCESS CREATE PROGRAM FILE ERROR.
```

Cause. A PROCESS_CREATE_ error occurred on the program file parameter.

Recovery. For the programmatic interface, call the SORTERRORDETAIL or SORTERRORSUM procedure to determine the error code. For the interactive interface, the *`error`* parameter in the accompanying message identifies the error code.

Follow the recovery recommendations in the *Guardian Procedure Errors and Messages Manual* for the error code.

```
161    PROCESS CREATE LIBRARY FILE ERROR.
```

Cause. A PROCESS_CREATE_ error occurred on the library file parameter.

Recovery. For the programmatic interface, call the SORTERRORDETAIL or SORTERRORSUM procedure to determine the error code. For the interactive interface, the *`error`* parameter in the accompanying message identifies the error code.

Follow the recovery recommendations in the *Guardian Procedure Errors and Messages Manual* for the error code.

```
162    PROCESS CREATE SWAP ERROR.
```

Cause. A PROCESS_CREATE_ error occurred on the swap file parameter.

Recovery. For the programmatic interface, call the SORTERRORDETAIL or SORTERRORSUM procedure to determine the error code. For the interactive interface, the *`error`* parameter in the accompanying message identifies the error code.

Follow the recovery recommendations in the *Guardian Procedure Errors and Messages Manual* for the error code.

```
163    PROCESS CREATE EXTENDED SWAP FILE ERROR.
```

Cause. A PROCESS_CREATE_ error occurred on the extended swap file parameter.

Recovery. For the programmatic interface, call the SORTERRORDETAIL or SORTERRORSUM procedure to determine the error code. For the interactive interface, the *`error`* parameter in the accompanying message identifies the error code.

Follow the recovery recommendations in the *Guardian Procedure Errors and Messages Manual* for the error code.

```
164    PROCESS CREATE DATA SEGMENT ERROR.
```

Cause. A PROCESS_CREATE_ error occurred for the process file segment (PFS).

Recovery. For the programmatic interface, call the SORTERRORDETAIL or SORTERRORSUM procedure to determine the error code. For the interactive interface, the *`error`* parameter in the accompanying message identifies the error code.

Follow the recovery recommendations in the *Guardian Procedure Errors and Messages Manual* for the error code.

```
165    PROCESS CREATE SYSTEM MONITOR ERROR.
```

Cause. A PROCESS_CREATE_ error occurred because the process could not communicate with the system monitor process.

Recovery. For the programmatic interface, call the SORTERRORDETAIL or SORTERRORSUM procedure to determine the error code. For the interactive interface, the *`error`* parameter in the accompanying message identifies the error code.

Follow the recovery recommendations in the *Guardian Procedure Errors and Messages Manual* for the error code.

```
166    PROCESS CREATE PROGRAM FILE FORMAT ERROR.
```

Cause. A PROCESS_CREATE_ error occurred because the program file has an invalid format.

Recovery. For the programmatic interface, call the SORTERRORDETAIL or SORTERRORSUM procedure to determine the error code. For the interactive interface, the *`error`* parameter in the accompanying message identifies the error code.

Follow the recovery recommendations in the *Guardian Procedure Errors and Messages Manual* for the error code.

```
167    PROCESS CREATE LIBRARY FILE FORMAT ERROR.
```

Cause. A PROCESS_CREATE_ error occurred because the program file has an invalid format.

Recovery. For the programmatic interface, call the SORTERRORDETAIL or SORTERRORSUM procedure to determine the error code. For the interactive interface, the *`error`* parameter in the accompanying message identifies the error code.

Follow the recovery recommendations in the *Guardian Procedure Errors and Messages Manual* for the error code.

```
168    INVALID STATISTICS LENGTH SPECIFIED.
```

Cause. The length parameter specified in the call to SORTMERGESTATISTICS was invalid.

Effect. The SQL DDL or DML operation in progress terminates abnormally.

Recovery. Correct the value of the parameter, then reissue the request.

```
169    INVALID STATISTICS FLAG VALUE SPECIFIED.
```

Cause. The value specified in the *`flags`* parameter to SORTMERGESTATISTICS was invalid.

Effect. The SQL DDL or DML operation in progress terminates abnormally.

Recovery. Specify a value of 0 or 1 for the parameter, then reissue the request.

```
170    SEGMENTS ABOVE 32767 NOT ALLOWED WITH VLM OFF.
```

Cause. You specified a SEGMENT value greater than 32,767 and have not requested the VLM option.

Effect. The SQL DDL or DML operation in progress terminates abnormally.

Recovery. Either specify the VLM option or specify a SEGMENT value less than or equal to 32,767.

```
171    EXTENDED SEGMENT CANNOT BE DEALLOCATED.
```

Cause. The SORTPROG process encountered an error while trying to deallocate its extended segment.

Effect. None; the problem occurs at process termination time.

Recovery. No recovery is necessary. However, you should report this error to your service provider.

```
172 SORTPROG VERSION TOO OLD; CANNOT SUPPORT REQUIRED NEW
FEATURE.
```

Cause. Your system’s version of SORTPROG is older than the FastSort system library procedures.

Effect. The SQL DDL or DML operation in progress terminates abnormally.

Recovery. Contact your system manager or service provider to have the correct version of FastSort or the operating system installed on your system.

```
173 ERROR IN MOVEX.
```

Cause. A programming error occurred on a call to an internal procedure.

Recovery. Report the internal error number returned with this FastSort error code to your service provider. The high-order word of the *`error-code`* parameter returned by SORTERRORSUM and SORTERRORDETAIL contains the internal error number.

```
174 MONITOR VERSION AND MESSAGE LENGTH CONFLICT.
```

Cause. Your system’s version of SORTPROG is older than the FastSort system library procedures.

Effect. The SQL DDL or DML operation in progress terminates abnormally.

Recovery. Contact your system manager or service provider to have the correct version of FastSort or the operating system installed on your system.

```
175 INVALID MONITOR MESSAGE LENGTH.
```

Cause. Your system’s version of SORTPROG is older than the FastSort system library procedures.

Effect. The SQL DDL or DML operation in progress terminates abnormally.

Recovery. Contact your system manager or service provider to have the correct version of FastSort or the operating system installed on your system.

```
176 SORTPROG VERSION TOO OLD; CANNOT SUPPORT OPTIONAL OPEN-
ON-DEMAND FEATURE.
```

Cause. Your system’s version of SORTPROG is older than the FastSort system library procedures.

Effect. The SQL DDL or DML operation in progress terminates abnormally.

Recovery. Contact your system manager or service provider to have the correct version of FastSort or the operating system installed on your system.

Alphabetic List of Interactive Messages

This subsection describes the interactive FastSort messages in alphabetic order. This description includes the error message text, the probable cause of the error, and the recommended recovery.

To determine appropriate recovery action for some errors, refer to the *Guardian Procedure Errors and Messages Manual*, which has information about the file-system and NEWPROCESS error codes that accompany FastSort error codes.

```
A THRU IS INCORRECT IN THE COLLATING SEQUENCE SPECIFICATION
```

Cause. The collating sequence you specified was invalid. For example, you specified Z THRU A instead of A THRU Z.

Recovery. Correct the collating sequence so that it follows the rules under the COLLATE Command on page 3-6.

```
AN ALSO MODIFIES A SPECIFIER WITH A DIFFERENT LENGTH.
```

Cause. You specified an incorrect collating sequence. The ALSO option indicates that two values are equal. For example, “A” ALSO “a” is a valid statement; however, “A” THRU “B” ALSO “a” is invalid because there is nothing to compare for B.

Recovery. Correct the command so that it follows the rules under the COLLATE Command on page 3-6.

```
ASCENDING n FOR n.
```

Cause. Information only.

```
CANNOT DO THE COLLATEOUT STATEMENT.
```

Cause. FastSort was unable to open or write the collate sequence to the output file specified.

Recovery. Use the accompanying error messages to determine what is wrong and correct the problem.

```
CANNOT INCREASE THE SCRATCH FILE SIZE
```

Cause. The SORTPROG process was unable to increase the maximum number of extents for a scratch file because one of the following errors occurred:

- There are no more overflow scratch volumes available to SORTPROG.

- This sort operation requires more than 32 scratch files, and an increase of the maximum number of extents for the last scratch file would cause the file to exceed 2 GB or 978 extents. FastSort also returns file system error 21 (ILLEGAL *count* SPECIFIED).
- A file-system error other than 21 occurred when SORTPROG tried to increase the number of extents for a scratch file.

Recovery. For the first error, use the SCRATCHON or NOSCRATCHON SORT DEFINE attribute to specify additional scratch volumes. For the second error, follow the recovery recommendations in the *Guardian Procedure Errors and Messages Manual* for the file-system error code returned with the FastSort error code.

```
CPUS cpu-list.
```

Cause. Information only.

```
CPUS ALL
```

Cause. Information only.

```
DEFAULT DEFINE IS NOT OF CLASS SORT.
```

Cause. The DEFINE =_SORT_DEFAULTS is not of class SORT.

Recovery. Specify CLASS SORT in your SORT DEFINEs.

```
DEFINE HAS BEEN SPECIFIED BUT DEFMODE IS OFF.
```

Cause. DEFMODE must be on to activate DEFINEs.

Recovery. Set DEFMODE to ON or determine why DEFMODE is not ON.

```
DESCENDING n FOR n.
```

Cause. Information only.

```
ERROR OCCURRED WHILE ACCESSING A SORT DEFINE.
```

Cause. Unacceptable DEFINE name encountered, attribute missing, or error from procedure call.

Recovery. Check that a valid DEFINE name was specified.

```
ERROR OCCURRED WHILE ACCESSING A SUBSORT DEFINE.
```

Cause. Unacceptable DEFINE name encountered, attribute missing, or error from procedure call.

Recovery. Check that a valid DEFINE name was specified.

```
FEATURE NOT SUPPORTED YET text.
```

Cause. Information only.

```
FILE NAME NOT SPECIFIED.
```

Cause. You omitted the file name.

Recovery. Specify the file name.

```
FROM filename
```

Cause. Information only.

```
IGNORING UNUSABLE STRING OF LETTERS text.
```

Cause. You mistyped the command or included text in the command that is not needed.

Recovery. Check the last command for errors and retype the command correctly.

```
INPUT FILE DOESN'T CONTAIN AN ENTIRE COLLATING SEQUENCE
TABLE.
```

Cause. You did not assign all characters a specific sequence. For example, you have specified “a - b - c - e - f” in order, but did not specify where “d” fits in.

Recovery. Correct the file so that all characters are assigned a sequence.

```
INTEGER CONVERSION ERROR.
```

Cause. You entered a number too big for FastSort to handle (for example, RUN, BLOCK 900000). This could also be an internal sort error message.

Recovery. Enter a valid value less than 32,767. If this is an internal sort error message, contact your service provider.

```
INVALID CHARACTER.
```

Cause. You used an incorrect character.

Recovery. Retype the command without the incorrect character.

```
INVALID FILE NAME.
```

Cause. The file name you specified was invalid.

Recovery. Check the file name to make sure it was correctly typed. See the *Guardian Programmer's Guide* for rules about specifying files.

```
INVALID OR NON-EXISTENT USER-SPECIFIED DEFINE NAME.
```

Cause. The DEFINE name you specified was not valid.

Recovery. Specify a valid DEFINE name.

```
INVALID SYNTAX text.
```

Cause. You used incorrect syntax.

Recovery. Check the syntax and correct the error.

```
MORE THAN ONE SUBSORT SHOULD BE SPECIFIED.
```

Cause. You specified only one subsort for a parallel sort.

Recovery. Specify at least two subsorts.

```
NO ALLOWED CPUS.
```

Cause. Information only.

```
NO ASCENDING OR DESCENDING STATEMENTS HAVE BEEN ISSUED.
```

Cause. You did not issue a valid ASCENDING or DESCENDING command for the sort or merge run.

Recovery. Specify the sort key order in at least one ASCENDING or DESCENDING command.

```
NO COLLATE STATEMENT HAS BEEN ISSUED.
```

Cause. Information only.

```
NO FORBIDDEN CPUS.
```

Cause. Information only.

```
NO FROM STATEMENTS HAVE BEEN ISSUED.
```

Cause. The command file did not include any FROM commands or input records.

Recovery. Put a FROM command or input records in the command file.

```
NO SUBSORT STATEMENTS HAVE BEEN ISSUED.
```

Cause. Information only.

```
NO TO STATEMENT HAS BEEN ISSUED.
```

Cause. Information only.

```
NOTCPUS cpu-list.
```

Cause. Information only.

```
ONLY ONE FROM STATEMENT MAY SPECIFY INPUT FROM THE COMMAND
FILE.
```

Cause. You issued too many FROM commands.

Recovery. Issue the command FROM *in-file* command only once. Note that if you want to sort records from both the terminal and from an input file, you can issue the FROM command once and the FROM *in-file* command once.

```
OUTPUT RECORD WOULD EXCEED BUFFER SPACE.
```

Cause. You specified an invalid record length.

Recovery. Correct the record length to a valid size.

```
SCRATCH FILE MUST BE UNSTRUCTURED.
```

Cause. A scratch file named in the call to SORTMERGESTART is a structured file.

Recovery. Specify an unstructured scratch file.

```
SUBSORT  , BLOCK block size.
```

Cause. Information only.

```
SUBSORT  , CPU cpu.
```

Cause. Information only.

```
SUBSORT DEFINE IS NOT OF CLASS SUBSORT.
```

Cause. The DEFINE class must be SUBSORT.

Recovery. Specify CLASS SUBSORT in your SUBSORT DEFINEs.

```
SUBSORT  , PRI priority.
```

Cause. Information only.

```
SUBSORT  , PROGRAM program name.
```

Cause. Information only.

```
SUBSORT  , SEGMENT segment.
```

Cause. Information only.

```
SUBSORT  , SWAP swap-file-name.
```

Cause. Information only.

```
SUBSORT scratch-file-name,
```

Cause. Information only.

```
THE ALTERNATE COLLATING SEQUENCE SPECIFICATION IS INCORRECT.
```

Cause. You specified the alternate collating sequence incorrectly.

Recovery. Correct the command so that it follows the rules under the COLLATE Command on page 3-6.

```
THE COLLATE FILE MUST BE UNSTRUCTURED.
```

Cause. The file specified in the COLLATE command is not unstructured.

Recovery. Specify an EDIT file (file code 101) or a file created in a previous COLLATEOUT command.

```
THE COLLATEOUT FILE MUST BE AN ENSCRIBE FILE.
```

Cause. The file specified in the COLLATEOUT command is an SQL object.

Recovery. Specify an unstructured Enscribe file in the COLLATEOUT command.

```
THE COLLATING SEQUENCE CANNOT BE DISPLAYED.
```

Cause. You entered SHOW COLLATE, which is not a valid command.

Recovery. If you have executed the COLLATEOUT command to store the collating sequence table in an unstructured file, you can view the file using the TACL VIEW command.

```
THE CPU NUMBER MUST NOT EXCEED 16.
```

Cause. You specified a processor (CPU) number larger than 16.

Recovery. Correct the command so that the number is less than 16.

```
THE MEMORY SIZE IS NOT IN RANGE 1-64.
```

Cause. You specified memory size not in the range from 1 to 64.

Recovery. Omit the MEM parameter of the RUN command.

```
THE PRIORITY IS NOT IN RANGE 1-199
```

Cause. You specified a priority not in the valid range.

Recovery. Correct the PRI parameter of the RUN command so that the priority is within the range from 1 to 199.

```
THE SCRATCH FILE BLOCK SIZE MAY NOT EXCEED 55296.
```

Cause. You used an invalid scratch file block size.

Recovery. Use a valid scratch file block size. The block size must be a multiple of 2048 up to 55296.

```
THE STARTING COLUMN MUST BE BEFORE THE END COLUMN.
```

Cause. You specified a sort key field whose starting column number is greater than the ending column number.

Recovery. Correct the ASCENDING or DESCENDING command so that the starting column number is less than the ending column number.

```
THE SYSTEM NAME IS UNRECOGNIZABLE.
```

Cause. You entered an invalid system name.

Recovery. Correct the SYSTEM parameter of the RUN command so that the node name is valid. A node name begins with a backslash (\) and is followed by a letter and up to 6 alphanumeric characters.

```
THE VALID KEY COLUMNS ARE 1 THROUGH 4080.
```

Cause. You specified a key outside the record.

Recovery. Correct the ASCENDING or DESCENDING command so that you specify a valid key field.

```
THERE IS ALREADY A DESTINATION FILE.
```

Cause. You specified more than one destination file.

Recovery. Clear the existing TO command or do not specify another one.

```
THIS PROGRAM CAN ONLY HANDLE 63 KEYS.
```

Cause. You specified too many keys in an ASCENDING or DESCENDING command.

Recovery. Retype the command and specify 63 or fewer keys.

```
TRUNCATING OUTPUT RECORD LENGTH TO DEVICE WIDTH.
```

Cause. The output file record size is too small to hold an output record.

Recovery. Create the output file with a larger record size and then rerun the sort.

```
USER-SPECIFIED DEFINE IS NOT OF CLASS SORT.
```

Cause. The DEFINE CLASS must be SORT.

Recovery. Specify CLASS SORT in your SORT DEFINEs.

```
WRONG NUMBER OF ELEMENTS IN SPECIFIER SEQUENCE, MUST BE 256.
```

Cause. You specified an invalid collating sequence.

Recovery. Correct the collating sequence to include 256 elements. Follow the rules under the COLLATE Command on page 3-6.

```
WRONG NUMBER OF SUBSORTS, MUST BE BETWEEN 2 AND 16.
```

Cause. You specified more than 16 subsorts.

Recovery. Clear the SUBSORT command and specify from 2 to 16 subsorts. Because using more than 8 subsorts can cause run-time errors, HP recommends that you specify a maximum of 8 subsorts. The number of actual subsorts you can use depends on your system configuration and load.

```
WRONG SEGMENT SIZE NUMBER, MUST BE LARGER THAN 64.
```

Cause. You specified an invalid segment size.

Recovery. Specify a segment size greater than 64 pages.

```
YOU MUST HAVE A LIST DEVICE WHEN REQUESTING OUTPUT THERE.
```

Cause. You specified a list device that does not exist.

Recovery. Check that the list device does exist and retype the command.

C Using Supported File Types

For input files, FastSort accepts records from unstructured, relative, entry-sequenced, key-sequenced, EDIT, and partitioned files.

FastSort does not accept input records from the following:

- Blocked tape files
- Key-sequenced files with increased limits

You might use buffered interface to send records from key-sequenced files with increased limits to FastSort. You do not specify the type for input files.

For output files, FastSort can create any type of output file except an EDIT file, and you can use an existing output file of any type that FastSort can create. If the output file exists, FastSort purges all data from the file before using it. If the output file is too small, FastSort purges it and re-creates the file. FastSort does not write records onto key-sequenced files with increased limits. For more information about key-sequenced files with increased limits, see *Enscribe Programmer's Guide*.

If you do not specify a file type for the output file, SORTPROG sets the type as follows:

- If the output file already exists, SORTPROG uses the type of the output file.
- If the output file does not already exist, SORTPROG uses the file type of the first input file.
- If the input records are from the SORTMERGESEND procedure, the output file type is entry-sequenced.

If you wish, however, you can specify the output file type as follows:

- For the interactive interface, set the TYPE parameter of the TO command.
- For the programmatic interface, set the SORTMERGESTART procedure *`out-file-type`* parameter.

Table C-1 summarizes the output file types.

Table C-1. Summary of Output File Types

Output File Type	TO Command TYPE Parameter	SORTMERGESTART Procedure *`output-file`* Parameter
Unstructured	U	0
Relative	R	1
Entry-sequenced	E	2

Table C-1. Summary of Output File Types

Output File Type	TO Command TYPE Parameter	SORTMERGESTART Procedure *output-file* Parameter
Key-sequenced	K	3
EDIT	Use EDIT to create the file and copy data from another file type.	–
Tape	Use FUP to load the file.	–

For information about the different types of files, see the *Guardian Programmer's Guide* and the *Enscribe Programmer's Guide*.

Unstructured Files

You can use unstructured files for input and output files; however, you cannot use an EDIT file, which is a special kind of unstructured file, as an output file. You can copy output records from an unstructured file to an EDIT file, as described under EDIT Files on page C-4.

For an unstructured output file, specify either of the following:

- U in the TYPE parameter of the TO command
- 0 (zero) in the *out-file-type* parameter of the SORTMERGESTART procedure

If the type of your first input file is unstructured, the default output file type is unstructured.

To use an odd unstructured file for an input file, you must specify the correct record length as follows:

- For the interactive interface, set the RECORD *length* parameter in the FROM command.
- For the programmatic interface, set the SORTMERGESTART *in-file-record-length* parameter.

To use an odd unstructured file for an output file, create the file using the FUP CREATE command or the CREATE system procedure before the sort or merge run. Then perform one of these steps:

- For the interactive interface, do not set the TYPE *file-type* parameter in the TO command.
- For the programmatic interface, set the SORTMERGESTART *out-file-type* parameter to –1.

Relative Files

You can use relative files as input or output files. For a relative output file, specify either of the following:

- R in the TYPE parameter of the TO command
- 1 in the *`out-file-type`* parameter of the SORTMERGESTART procedure

If the type of your first input file is relative, the default output file type is relative.

Entry-Sequenced Files

You can use entry-sequenced files as input and output files. For an entry-sequenced output file you do not need to specify the type because entry-sequenced is the default type (unless the first input file is not an entry-sequenced file). However, you can specify either of the following:

- E in the TYPE parameter of the TO command
- 2 in the *`out-file-type`* parameter of the SORTMERGESTART procedure

If the type of your first input file is entry-sequenced, the default output file type is entry-sequenced.

Key-Sequenced Files

You can use key-sequenced files as input files and output files. No special requirements apply to using key-sequenced input files.

If you use a key-sequenced output file, you can specify only one key field for sorting. That field must be the same as the primary key field for the file. When using commands, you must name the field in an ASCENDING command and specify UNSIGNED as the data type. When using the SORTMERGESTART procedure, you must specify ascending and BINARY UNSIGNED for the field in the *`key-block`* parameter array.

To cause FastSort to create a key-sequenced output file or use an existing one, you must specify the type. You can also specify the percentage of slack space for accommodating future insertions of records in a new or existing key-sequenced file.

To specify the output file type, use either of the following:

- K in the TYPE parameter of the TO command
- 3 in the *`out-file-type`* parameter of the SORTMERGESTART procedure

To specify the data and index slack, use any of the following:

- The SLACK parameter of the RUN command, if you want the same percentage of slack space in the data blocks and the index blocks

- The DSLACK and ISLACK parameters of the RUN command, if you want the data blocks to have a different percentage of slack space than the index blocks
- The *dslack* and *islack* parameters of the SORTMERGESTART procedure

The default for SLACK, *dslack*, and *islack* is 0 percent. The default for DSLACK and ISLACK is the value of SLACK.

FastSort currently does not load alternate-key files directly. You can use FUP to load alternate-key files. For information about loading alternate-key files, see the *Guardian User's Guide*.

If the type of your first input file is key-sequenced, the default output file type is entry-sequenced.

FastSort currently supports key-sequenced files with increased limits only through buffered interface. For more information about key-sequenced files with increased limits, see *Enscribe Programmer's Guide*.

EDIT Files

FastSort accepts EDIT (file code 101) files as input files but not as an output file. If the type of your first input file is EDIT, the default output file type is entry-sequenced. If you want output records from a sort or merge run in an EDIT file, you must copy the output records into an existing or new EDIT file.

First, use FastSort to sort the records to a structured output file. Then, use EDIT to copy the output records from the structured output file to a new or existing EDIT file. For example, the following sequence of commands, entered at a TACL prompt, copies the sorted records from the SORTOUT file to an EDIT file named NEWFILE:

```
EDIT NEWFILE !; GET SORTOUT TO LAST; EXIT
```

The exclamation point (!) causes EDIT to create NEWFILE without prompting you for confirmation if the file does not exist. The LAST parameter causes EDIT to write the records from SORTOUT after the last line in NEWFILE. You can also specify a line number rather than LAST to have EDIT insert the records after that line. For example, the following sequence of commands inserts the sorted records after line 1 in NEWFILE.

```
EDIT NEWFILE !; GET SORTOUT TO 1; EXIT
```

Any existing data in NEWFILE remains in the file after the sorted records. If you need to first purge data from an existing EDIT file, use the FUP PURGEDATA command.

For more information about using EDIT commands, see the *EDIT User's Guide and Reference Manual.*

Tape Files

If you want to use input records from a blocked tape file, use FUP to deblock the records by loading them into a disk file. Then specify the disk file as an input file for the sort or merge run.

If you want to store output records in a blocked tape file, use FUP to block the records by loading them to the tape file from a disk file. Then specify the disk file as the output file for the sort or merge run.

For information about using the FUP LOAD command, see the *FUP Reference Manual*.

The COBOL85 SORT and MERGE statements use FastSort to deblock and block tape files for you. For a description of the SORT and MERGE statements, see the *COBOL85 Reference Manual*.

Partitioned Files

FastSort accepts partitioned input files and can write records to a partitioned output file. FastSort can also use a partitioned scratch file. A partitioned file, however, must exist before you use it as an input, output, or scratch file. To create a partitioned file, use FUP. For more information on how to create a partitioned scratch file, see Section 9, Optimizing Sort Performance.

Partitioned Output Files

Use a partitioned output file for a distributed database or for a set of output records that will not fit on one disk volume. To estimate the size of the output file, multiply the total number of input records by the maximum output record length.

If one or more input files is partitioned, you do not need to use a partitioned output file, unless the output records will not fit on one disk volume. Output records from permutation sorts are shorter than output records from record sorts, and output records from key sorts can be even shorter. Also, if you have FastSort remove records with duplicate key values, the output records from a record sort usually take up less space than the input records.

If FastSort determines that an output file is too small, it purges and re-creates the file. For a partitioned output file, however, you can prevent FastSort from purging and re-creating the file in order to preserve the original partitioning and extents of the file. To prevent FastSort from purging the file, specify the NOPURGE parameter of the TO command or set the SORTMERGESTART procedure *flags*.<5> bit to 1.

D ASCII Character Set

Table D-1. ASCII Character Set (page 1 of 4)

Character	Octal Left	Octal Right	Hex	Dec	Meaning
NUL	000000	000000	00	0	Null
SOH	000400	000001	01	1	Start of heading
STX	001000	000002	02	2	Start of text
ETX	001400	000003	03	3	End of text
EOT	002000	000004	04	4	End of transmission
ENQ	002400	000005	05	5	Enquiry
ACK	003000	000006	06	6	Acknowledge
BEL	003400	000007	07	7	Bell
BS	004000	000010	08	8	Backspace
HT	004400	000011	09	9	Horizontal tabulation
LF	005000	000012	A	10	Line feed
VT	005400	000013	B	11	Vertical tabulation
FF	006000	000014	C	12	Form feed
CR	006400	000015	D	13	Carriage return
SO	007000	000016	E	14	Shift out
SI	007400	000017	F	15	Shift in
DLE	010000	000020	10	16	Data link escape
DC1	010400	000021	11	17	Device control 1
DC2	011000	000022	12	18	Device control 2
DC3	011400	000023	13	19	Device control 3
DC4	012000	000024	14	20	Device control 4
NAK	012400	000025	15	21	Negative acknowledge
SYN	013000	000026	16	22	Synchronous idle
ETB	013400	000027	17	23	End of transmission block
CAN	014000	000030	18	24	Cancel
EM	014400	000031	19	25	End of medium
SUB	015000	000032	1A	26	Substitute
ESC	015400	000033	1B	27	Escape
FS	016000	000034	1C	28	File separator
GS	016400	000035	1D	29	Group separator
RS	017000	000036	1E	30	Record separator
US	017400	000037	1F	31	Unit separator

Table D-1. ASCII Character Set (page 2 of 4)

Character	Octal Left	Octal Right	Hex	Dec	Meaning
SP	020000	000040	20	32	Space
!	020400	000041	21	33	Exclamation point
"	021000	000042	22	34	Quotation mark
#	021400	000043	23	35	Number sign
$	022000	000044	24	36	Dollar sign
%	022400	000045	25	37	Percent sign
&	023000	000046	26	38	Ampersand
'	023400	000047	27	39	Apostrophe
(	024000	000050	28	40	Opening parenthesis
)	024400	000051	29	41	Closing parenthesis
*	025000	000052	2A	42	Asterisk
+	025400	000053	2B	43	Plus
,	026000	000054	2C	44	Comma
-	026400	000055	2D	45	Hyphen (minus)
.	027000	000056	2E	46	Period (decimal point)
/	027400	000057	2F	47	Slash
0	030000	000060	30	48	Zero
1	030400	000061	31	49	One
2	031000	000062	32	50	Two
3	031400	000063	33	51	Three
4	032000	000064	34	52	Four
5	032400	000065	35	53	Five
6	033000	000066	36	54	Six
7	033400	000067	37	55	Seven
8	034000	000070	38	56	Eight
9	034400	000071	39	57	Nine
:	035000	000072	3A	58	Colon
;	035400	000073	3B	59	Semicolon
<	036000	000074	3C	60	Less than
=	036400	000075	3D	61	Equals
>	037000	000076	3E	62	Greater than
?	037400	000077	3F	63	Question mark
@	040000	000100	40	64	Commercial at sign
A	040400	000101	41	65	Uppercase A

Table D-1. ASCII Character Set (page 3 of 4)

Character	Octal Left	Octal Right	Hex	Dec	Meaning
B	041000	000102	42	66	Uppercase B
C	041400	000103	43	67	Uppercase C
D	042000	000104	44	68	Uppercase D
E	042400	000105	45	69	Uppercase E
F	043000	000106	46	70	Uppercase F
G	043400	000107	47	71	Uppercase G
H	044000	000110	48	72	Uppercase H
I	044400	000111	49	73	Uppercase I
J	045000	000112	4A	74	Uppercase
K	045400	000113	4B	75	Uppercase K
L	046000	000114	4C	76	Uppercase L
M	046400	000115	4D	77	Uppercase M
N	047000	000116	4E	78	Uppercase N
O	047400	000117	4F	79	Uppercase O
P	050000	000120	50	80	Uppercase P
Q	050400	000121	51	81	Uppercase Q
R	051000	000122	52	82	Uppercase R
S	051400	000123	53	83	Uppercase S
T	052000	000124	54	84	Uppercase T
U	052400	000125	55	85	Uppercase U
V	053000	000126	56	86	Uppercase V
W	053400	000127	57	87	Uppercase W
X	054000	000130	58	88	Uppercase X
Y	054400	000131	59	89	Uppercase Y
Z	055000	000132	5A	90	Uppercase Z
[	055400	000133	5B	91	Opening bracket
\	056000	000134	5C	92	Backslash
]	056400	000135	5D	93	Closing bracket
^	057000	000136	5E	94	Circumflex
_	057400	000137	5F	95	Underscore
`	060000	000140	60	96	Grave accent
a	060400	000141	61	97	Lowercase a
b	061000	000142	62	98	Lowercase b
c	061400	000143	63	99	Lowercase c

Table D-1. ASCII Character Set (page 4 of 4)

Character	Octal Left	Octal Right	Hex	Dec	Meaning
d	062000	000144	64	100	Lowercase d
e	062400	000145	65	101	Lowercase e
f	063000	000146	66	102	Lowercase f
g	063400	000147	67	103	Lowercase g
h	064000	000150	68	104	Lowercase h
i	064400	000151	69	105	Lowercase i
j	065000	000152	6A	106	Lowercase j
k	065400	000153	6B	107	Lowercase k
l	066000	000154	6C	108	Lowercase l
m	066400	000155	6D	109	Lowercase m
n	067000	000156	6E	110	Lowercase n
o	067400	000157	6F	111	Lowercase o
p	070000	000160	70	112	Lowercase p
q	070400	000161	71	113	Lowercase q
r	071000	000162	72	114	Lowercase r
s	071400	000163	73	115	Lowercase s
t	072000	000164	74	116	Lowercase t
u	072400	000165	75	117	Lowercase u
v	073000	000166	76	118	Lowercase
w	073400	000167	77	119	Lowercase w
x	074000	000170	78	120	Lowercase x
y	074400	000171	79	121	Lowercase y
z	075000	000172	7A	122	Lowercase z
{	075400	000173	7B	123	Opening brace
|	076000	000174	7C	124	Vertical line
}	076400	000175	7D	125	Closing brace
~	077000	000176	7E	126	Tilde
DEL	077400	000177	7F	127	Delete

E FastSort Limits

This appendix summarizes parameter values that FastSort accepts in commands and procedure calls.

Table E-1. FastSort Limits

Item	**Limit**
CPUs	0 to 15
Input Files/Streams	32
Key Columns (Non-SQL)	1 to 4080
Key Fields	1 to 63
Memory	1 to 64 pages
Priority	1 to 199
Segment (Extended Memory)	256 to 62,255 pages (with VLM on)
Subsort Processes	2 to 16 (No more than 8 recommended)
Scratch File Block Size	Any multiple of 2048 up to 56 kilobytes

Glossary

alternate collating sequence. An EDIT file instructs FastSort to collate sort results by specific alphanumeric key fields or string type data. The default FastSort collating sequence is the ASCII character set.

application . One or more processes that achieve a specific objective. Processes in an application often communicate with each other using the message system and file system. See also program and process.

command file. An EDIT file containing a sequence of commands to execute. When you execute the file, commands in the file are automatically executed. You can use a command file with FastSort to execute commands or set DEFINEs.

data stack space. A storage area for object files. Data stack space is automatically allocated and can be manually specified for a program either at compile or bind time.

file system. A set of operating procedures and data structures that allows communication between a process and a file, which can be a disk file, I/O device, or another process.

initial scratch file. The scratch file FastSort uses first to store partial information during a sort-merge operation. See also scratch file and overflow scratch file.

input file. A set of records from local or remote disk files, tape files, or a terminal that you specify to FastSort to sort or merge. Supported types of output files for FastSort are unstructured, relative, entry-sequenced, key-sequenced, and EDIT.

list file. The file FastSort creates after a sort or merge run that describes the run. For example, a list file can contain FastSort statistics and any errors or warnings that occurred during the run. By default, the list file is the home terminal for the FastSort process; it can also be a disk file, I/O device, SPOOL DEFINE, or a process that receives sort-merge output. See also SPOOL DEFINE.

output file. The file to which FastSort writes output records. By default, the output file is the home terminal for the FastSort process. Supported output file types for FastSort are unstructured, relative, entry-sequenced, and key-sequenced.

overflow scratch file. I f the initial scratch file becomes full during a sort-merge operation, the file FastSort creates to store overflow information. If there is sufficient overflow space, FastSort creates overflow scratch files on the same volume as the initial scratch file. See also scratch file and initial scratch file.

parallel sort-merge operation. A FastSort operation that improves performance by using one distributor-collector SORTPROG process and 2 to 8 subsort processes to distribute the sort workload to multiple processors.

process. An executing or running program that has been submitted to the operating system for execution.

program. A static set of instruction codes and initialized data, such as compiler output or the Binder program, that is not currently executing. A program usually resides in a program file on disk.

scratch file. A temporary work file for FastSort. When a sort-merge operation cannot be performed in memory, SORTPROG temporarily stores partial information in one or more scratch files.

serial sort-merge operation. A FastSort operation that uses one SORTPROG process to sort or merge records.

SORT DEFINE. An operating system parameter that affects sort operations. A DEFINE has a name and a set of attribute-value pairs. The =_SORT_DEFAULTS DEFINE is also a SORT DEFINE.

SORT process. The FastSort command interpreter process. SORT accepts interactive commands from a terminal or through a command file and then uses FastSort system procedures to send commands to a SORTPROG process. See also SORTPROG process.

SORTPROG process . The FastSort sort-merge process. SORTPROG can run as a single process for a serial sort-merge process or as a distributor-collector process with 2 to 8 subsort processes for a parallel sort-merge operation.

SPOOL DEFINE. An operating system parameter that affects output. You can specify a SPOOL DEFINE to receive sort-merge output. See also list file.

SUBSORT DEFINE. An operating system parameter that affects subsort processes in a parallel sort operation. A DEFINE has a name and a set of attribute-value pairs.

swap file. The disk file FastSort uses for data swapping during a sort or merge run. Data swapping involves copying data between physical memory and storage.

system message. A block of information, usually in the form of a structure, that a system process sends to another process. The receiving process, often a user process, reads system messages from the $RECEIVE system file. For example, an application that calls the FastSort SORTMERGEFINISH procedure with the *abort* parameter set to 1 (which means stop the SORTPROG process immediately) receives a process-deletion message in its $RECEIVE file. See also $RECEIVE.

$RECEIVE. A special system file through which a process receives and can reply to messages from other processes.

=_SORT_DEFAULTS DEFINE. The default SORT DEFINE. You can use the =SORT_DEFAULTS_DEFINE to specify FastSort parameters for applications that otherwise cannot set the parameters. For example, the =SORT_DEFAULTS_DEFINE affects all sort operations invoked by NonStop SQL/MP. See also SORT DEFINE.

Index