Burroughs

BTOS Sort/Merge

Operations Reference Manual

Relative to Release Level 7.0
Priced Item
January 1986

Distribution Code SA
Printed in U S America
5022148

Burroughs

BTOS Sort/Merge

Operations Reference Manual

Previous Title: B 20 Systems Sort/Merge Reference Manual

Relative to Release Level 7.0
Priced Item
January 1986

Distribution Code SA
Printed in U S America
5022148

Burroughs cannot accept any financial or other responsibilities that may be the result of your use of this information or software material, including direct, indirect, special or consequential damages. There are no warranties extended or granted by this document or software material.

You should be very careful to ensure that the use of this software material and/or information complies with the laws, rules, and regulations of the jurisdictions with respect to which it is used.

The information contained herein is subject to change without notice. Revisions may be issued to advise of such changes and/or additions.

Comments or suggestions regarding this document should be submitted on a Field Communication Form (FCF) with the CLASS specified as 2 (S.SW:System Software), the Type specified as 1 (F.T.R.), and the product specified as the 7-digit form number of the manual (for example, 5022148)

Contents

Introduction

This manual provides descriptive and operational information for the Burroughs Sort/Merge utility used in Burroughs applications.

Related Materials

For detailed information on the Burroughs Operating System (BTOS), refer to your *BTOS Reference Manual*.

For detailed information on Executive level commands, refer to your *BTOS Standard Software Operations Guide*.

For information on the Editor, refer to your *BTOS Editor Operations Guide*.

In addition, the following technical manuals are referenced in this manual:

- *BTOS Customizer Programming Reference Manual*
- *BTOS BASIC Compiler Programming Reference Manual*
- *BTOS FORTRAN Compiler Programming Reference Manual*
- *BTOS Indexed Sequential Access Method (ISAM) Operations Reference Manual*
- *BTOS Linker/Librarian Programming Reference Manual*

Conventions Used in this Manual

You must type items in uppercase letters in the order shown. You can enter them in either uppercase or lowercase. For example:

```
$END
```

Items in lowercase letters are variable information that you supply. For example:

```
$LOG 'message'
```

Items in square brackets are usually optional information. You do not type the brackets. For example:

```
$JOB jobname,username[,password][,SysOutfile]
```

Note, however, that you must type square brackets in full file specifications (refer to the last example) and in device names. For example:

```
[Kbd]
```

You type all punctuation (except square brackets around optional items) as shown. For example:

```
$JOB jobname,username,password
```

Where indicated, the full file specification for an abbreviated file specification, such as File0.Run, is:

```
{node}[vol]<dir>File0.Run
```

Overview

The Burroughs Sort/Merge facility is a system software product that sorts and merges data. Sort/Merge arranges a sequence of data records into a sorted sequence, or merges several sequences of sorted records into a single sorted sequence.

Sort/Merge consists of:

- an interactive Sort utility
- an interactive Merge utility
- key-in-record sort procedures
- external-key sort procedures

Sort/Merge Features

All the components of Sort/Merge support variable-length records and fixed-length keys. Sort/Merge supports sorts with a composite sort key put together either by the application program or by Sort/Merge, using key-in-record sort procedures.

Sort/Merge allows flexible specification of the sort key; it can be composed of multiple fields of a record with each field designated ascending or descending.

In addition, the interactive Sort/Merge utilities are distributed in both Run file and object module format. The latter format allows you to tailor the utilities through the addition of special user-written procedures (see sections 3 and 4).

Sort/Merge makes efficient use of the Burroughs Operating System (BTOS) capabilities by employing all available workstation memory as well as auxiliary disk files in its procedures.

Sort and Merge Utilities

The interactive Sort and Merge utilities sort or merge records contained in Standard Access Method (STAM) files. Direct Access Method (DAM) and Indexed Sequential Access Method (ISAM) use STAM files for fixed length records, and Record Sequential Access Method (RSAM) for variable length records.

The *BTOS Reference Manual* describes these file access methods. Also see the *BTOS Standard Software Operations Guide*.

The Sort utility accepts several files of unsorted records and sorts and merges the records to create a single output file.

The Merge utility accepts several files of sorted data records and merges them into a single sorted output file.

You activate the Sort and Merge utilities from the Executive as described in sections 3 and 4.

Object Module Procedures

The Sort/Merge object module procedures consist of key-in-record sort procedures and external-key sort procedures. You can link them into an application system and call them from many programming languages, such as BASIC, COBOL (which uses the COBOL Sort verb), FORTRAN, and Pascal.

When you use key-in-record sort procedures, the application program presents a single formula for extracting the sort key from each data record. The application program releases only data records, since the associated keys are extracted from the records automatically.

When you use external-key sort procedures, the application program must specify the sort key for each record as it is released to the sort.

Concepts

You decide the order in which you want records to be sorted and enter this parameter into the Sort/Merge facility.

Consider the records:

City	Population
Brigham	5,641
Logan	11,868
Murray	5,740
Ogden	43,688
Price	5,214
Provo	18,071
Salt Lake City	149,934
South Salt Lake	5,701
Tooele	5,001

As shown, the records are properly sorted in ascending alphabetical order by city. They could also be sorted in descending alphabetical order, and in ascending or descending numerical order by population.

All records have values that the system compares to determine their proper order. These values are called sort keys.

In the preceding example, the sort keys are Brigham, Logan, Murray, etc. If the same records were sorted in descending numerical order by population, the sort keys would be 149,934, 43,688, 18,071, etc., and the sorted records would be:

City	Population
Salt Lake City	149,934
Ogden	43,688
Provo	18,071
Logan	11,868
Murray	5,740
South Salt Lake	5,701
Brigham	5,641
Price	5,214
Tooele	5,001

Key Types

To allow most data representations specified in each programming language to be used as keys, Sort/Merge supports 12 types of keys.

A brief description of each key type follows. For more information on the relationships between key types and programming language representations, see table 5-3.

Binary

A binary key is an unsigned 1- to 8-byte integer. The high-address byte of a binary key is the most significant for determining sort order. For COBOL COMP fields, the low-address byte is the most significant.

Byte String

A byte string key is an uninterpreted fixed-length string of 1 to 64 binary bytes. The low-address bytes of the string are the most significant for determining sorting order. A distinction is made between uppercase and lowercase ASCII characters. Byte strings have the same representation in all programming languages.

Character String

A character string key is a fixed-length string of 1 to 64 binary bytes. Like a byte string, a character string is sorted with the low-address byte as the most significant. However, unlike a byte string, character string keys are sorted with no distinction between uppercase and lowercase ASCII characters. Character strings have the same representation in all programming languages.

Decimal (Odd)/Decimal (Even)

A decimal key contains two decimal digits in each byte, except for the last (high-address) byte, where the rightmost four bits are reserved for a sign. This format is the same as COBOL COMP-3.

Decimal (even) is used for values that have an even number of digits; decimal (odd) is used for values that have an odd number of digits. The number of digits before the number is packed determines whether the (even) or (odd) decimal type is used.

A decimal key can contain 1 to 18 decimal digits. Decimal fields have the same representation in all programming languages. For more information about this type of field, see your COBOL documentation.

Display

A display key is used in COBOL applications for USAGE IS DISPLAY numeric fields. All COBOL sign options are supported. Display keys can be 1 to 19 bytes long and contain 1 to 18 decimal digits. For more information about the range of values and representations for display keys, see your COBOL documentation.

Integer

An integer key is a signed 1- to 8-byte integer. The high-address byte of an integer key is the most significant for determining sort order. However, for COBOL COMP fields, the low-address byte is the most significant.

Long/Short/Extended IEEE

Long IEEE, short IEEE, and extended IEEE keys are used for real numbers in Pascal or FORTRAN applications. The high-address byte is the most significant byte for determining sort order.

A long IEEE key is 8 bytes long, a short IEEE key is 4 bytes long, and an extended IEEE key is 10 bytes long.

Long/Short Real

Long real and short real keys are used in BASIC applications. A long real key is an 8-byte real number; a short real key is a 4-byte real number.

For information regarding the number of bits of precision and range of values for these keys, see your COBOL documentation.

Multilevel Sort Capabilities

You can form a sort key by combining several parts of the record. Sort/Merge does multilevel sorts and keeps track of which components of the composite key are sorted in ascending order and which are sorted in descending order. For example, consider the records:

Part Number	Backlog
98-374	100
97-392	200
93-495	200
94-592	100

Suppose you want to sort these in descending order by backlog and, for records with the same backlog, in ascending order by part number. The results of this sort example are:

Part Number	Backlog
93-495	200
97-392	200
94-592	100
98-374	100

The external-key object module procedures do not support composite keys. The application system provides a single key with each record.

Merging

The Merge utility merges copies of several existing files and writes the merged records into a new Standard Access Method (STAM) file. The original files are untouched. For example, if one file contains the records:

City	Population
Salt Lake City	149,934
Provo	18,071
Logan	11,868
South Salt Lake	5,701
Brigham	5,641

and another file contains the records:

City	Population
Ogden	43,688
Murray	5,740
Price	5,214
Tooele	5,001

the results of merging these files in descending order by population are:

City	Population
Salt Lake City	149,934
Ogden	43,688
Provo	18,071
Logan	11,868
Murray	5,740
South Salt Lake	5,701
Brigham	5,641
Price	5,214
Tooele	5,001

Sort Utility

The interactive Sort utility is a part of the Sort/Merge facility that you activate directly from the Executive. It sorts preexisting files of data records according to sort keys embedded within those records.

The files can be any STAM files. You can create files with RSAM or DAM, or they can be the data store file of an ISAM data set. In ISAM, the result of the sort is a file that is accessible with RSAM or DAM, but is not a new ISAM data set. If you wish to create a new ISAM data set, consult the *BTOS Indexed Sequential Access Method (ISAM) Operations Reference Manual*.

Sort has special features to deal with input files that might contain malformed records. These features are described later in this section.

Activating Sort

To activate SORT from the Executive, you type **Sort** in the command field of the Executive command form and then press **RETURN**. The following form is displayed:

```
Sort
  Input files                  _____
  Output file                  _____
  Keys                         _____
  [Stable sort?]               _____
  [Work file 1]                _____
  [Work file 2]                _____
  [Log file]                   _____
  [Suppress confirmation?]     _____
```

You must fill in the first three fields. The remaining five fields are optional. You specify the default in an optional field by leaving it blank. After you have filled in the appropriate fields, you press **GO**.

Field Descriptions

Following are descriptions of each field that appears when you activate Sort:

- Input files specifies a list of the names of one or more files to be sorted. Separate the names with spaces, not commas. Each file must be a STAM file. All valid records in these files are sorted; deleted records are skipped. When Sort detects a malformed record, it activates the error handling facilities described later in this section.
- Output file specifies the name of the file to which Sort writes the sorted output. The output file is written with RSAM. However, if all of the input records have the same size, the output file is accessible with either DAM or RSAM.
- Keys specifies how sort keys are embedded within each data record. Although the input records can be of varying lengths, all must have a prefix of common fixed length containing the sort keys.

  If you want a multilevel sort, you must enter several specifications in the Keys field. Each specification represents one component of the sort key. Separate the specifications with spaces, not commas. If there is more than one specification, the ones that appear first are more significant in determining sort order than the ones that appear later.

  Each key component specification has the form:

  TypeName:Length.Offset.AorD.WorM

  TypeName specifies the internal representation of the key component. It is one of the following strings: Binary, Byte, Character, Decimal, Integer, LongReal, ShortReal, LongIEEE, ShortIEEE, ExtendedIEEE real, and Display. Capitalization is not significant (for example, shortreal and SHORTreal are equivalent).

Also, you can use any unique abbreviation instead of a fully spelled TypeName (for example, C or Char for Character). The meanings of these key types are:

- Binary: the key component is a 1- to 8-byte unsigned number. The colon and length following the TypeName are optional. The default is 2.
- Byte: the key component is a sequence of binary bytes of length specified by Length. The first byte is the most significant.
- Character: the key component is a sequence of text characters of length specified by Length. For purposes of sorting, lowercase alphabetic characters (61h through 7Ah) are mapped to the corresponding uppercase alphabetic characters (41h through 5Ah). Thus, **a** is equivalent to the letter **A**. The first byte is the most significant.
- Decimal: the key component is a packed decimal number in the format used by COBOL COMP-3 numeric data items. The number of digits in the packed decimal number is specified by length and must be in the range 1 through 18.
- LongReal: the key component is an 8-byte real number used by BASIC. You must omit the colon and Length following this TypeName.
- ShortReal: the key component is a 4-byte real number used by BASIC. You must omit the colon and Length following this TypeName.
- Integer: the key component is a 1- to 8-byte signed number. The colon and length following the TypeName are optional. The default is 2.
- Display: the key component can be 1 to 19 bytes long and is used in COBOL applications for USAGE IS DISPLAY numeric fields. For the range of values and representations for display keys, see your COBOL documentation.
- Long IEEE: the key component is an 8-byte real number used by all programming languages except BASIC. (However, Long, Short, and Extended IEEE numbers do not work with COBOL, which has no real numbers.)

- ShortIEEE: the key component is a 4-byte real number used by all programming languages except BASIC.
- ExtendedIEEE: the key component is a 10-byte real number used by all programming languages except BASIC.

Length specifies the length of the key component as a positive decimal number. This number is interpreted according to the TypeName it modifies, as described earlier.

Offset specifies a decimal number representing the relative byte position of the key component within a data record. For example, an offset of 0 means that the key component starts at the beginning of the record.

AorD specifies the order in which you want merged records arranged. A specifies that the records be arranged so that this key component is in ascending order. D specifies that the records be arranged so that this key component is in descending order.

Sort order is determined according to the type of key component. Thus, negative real numbers are understood to be smaller than positive real numbers; negative packed decimal numbers are understood to be smaller than positive packed decimal numbers.

As an example, suppose the records to be sorted have the form:

Offset	Field	Length	Type
0	Name	18 bytes	Character
18	Address	80 bytes	Character
98	Category	2 bytes	Binary
100	Identification Number	8 digits	Decimal

To sort these records in ascending order by Name, and descending order by Identification Number, enter the following in the Keys field:

```
Character : 18. 0. A
Decimal : 8. 100. D
```

To sort these records in descending order by Identification Number, ascending order by Category, and ascending order by Name, enter the following in the Keys field:

```
Decimal : 8. 100. D
Binary : 2. 98. A
Character : 18. 0. A
```

WorM specifies computer language application programs. Enter **W** for programs written to run in BASIC, FORTRAN, and Pascal. Enter **M** for programs written in COBOL. The default is W.

- [Stable sort?] specifies whether you want a stable sort. The default is No.

  Enter **Yes** for stable sort. A sort is said to be stable if input records whose sort keys are equal always appear in the output in the same order as they appear in the input.

  You should specify a stable sort only if one is necessary, since a stable sort takes longer to complete.

- [Work file 1] and [Work file 2] specify the names of two files Sort will use as work files. Sort requires a pair of work files, each approximately the same size as the input data. If you specify files that already exist, Sort uses these files and returns them at the end of the sort. If you specify files that do not exist, Sort creates them and deletes them at the end of the sort.

  If you do not specify work file names (the default), the work files are placed on the logged-in volume and directory and named SortWorkfile1.Dat and SortWorkfile2.Dat.

  For an efficient sort, you should make these work files physically contiguous and place them on different physical volumes. To make a file physically contiguous, you either create it when the disk is not very full or, after the file exists and has its maximum length, you use the **BACKUP VOLUME**, **IVOLUME**, and **RESTORE** commands to make all files physically contiguous. For a description of these commands, see the *BTOS Standard Software Operations Guide*.

- [Log file] specifies the name you choose for the file to which the status report and sort statistics are to be written.

Sort computes the following statistics and writes them to the log file: number of records, number of bytes of data, number of merge passes, and elapsed time.

If you do not specify a log file (the default), Sort will not produce one. However, all sort statistics and status codes display when the sort is complete.

- [Suppress confirmation?] specifies your desire to monitor the handling of malformed records by Sort. When Sort encounters malformed records in the input file, it displays a descriptive status code and writes it to the log file (if you have specified one).

  If you enter **Yes**, Sort automatically skips the malformed input and searches forward in the input data for the next well-formed record.

  If you enter **No**, Sort automatically skips the malformed input and displays a message that asks you whether you want the sort to continue or to terminate.

  However, you have an alternative to this method of error handling. Sort is supplied not only as a Run file but also as a library of object modules. You can tailor error handling to the requirements of your application by entering user-written procedures in place of the error handling module, as described later in this section.

Customizing Sort

The Sort utility is designed to call certain procedures in such a way that the application programmer can customize Sort by replacing these procedures with user-written code.

User-written code is code that you activate to preprocess all input records, postprocess all output records, and provide special error handling.

The library of Sort object modules, SortMerge.Lib, includes standard definitions for the following replaceable procedures:

- SortInStart
- SortIn
- SortInDone
- SortOutStart
- SortOut
- SortOutDone
- SortError

Sort controls the flow of the Sort operation by calling:

1. SortInStart once at the beginning of the sort
2. SortIn for each input record in the order read. (If malformed records are found, SortError is called instead of SortIn.)
3. SortInDone once after SortIn has been called for all the input records
4. SortOutStart once after the actual sort is complete
5. SortOut for each output record in sorted order
6. SortOutDone once after SortOut has been called for all the output records

Processing Input Records

SortInStart

The SortInStart procedure is called once at the beginning of the sort. It has the interface:

```
SortInStart: ErcType
```

This procedure has no parameters. The standard SortInStart is null; it does no work and returns immediately. However, you can substitute a custom version for the standard version to add initialization logic.

SortIn

The SortIn procedure is called for each input record in the order read.

The standard SortIn procedure included in SortMerge.Lib calls ReleaseRecord (described in section 6) on its input record, thus passing the input record into the standard Sort utility. To include user-written code for preprocessing input records, you build Sort with your own SortIn procedure that has the interface:

```
SortIn (pRecord, sRecord, iFile): ErcType
```

where

pRecord and sRecord describe the input record to be sorted.

iFile specifies the index of the input file within the specified list of input files (counted from zero for the first file).

The SortIn procedure can modify, delete, or insert input records. You modify input records by passing to ReleaseRecord a record different from the one with which it was called (see section 6 for a description of ReleaseRecord). Delete input records by returning to the calling procedure without calling ReleaseRecord for selected records. Insert input records by calling ReleaseRecord more than once on the basis of some computation.

SortInDone

The SortInDone procedure is called once after SortIn has been called for all the input records. It has the interface:

```
SortInDone: ErcType
```

This procedure has no parameters. The standard SortInDone is null. You can substitute a custom version of the standard version to add termination logic.

Processing Output Records

SortOutStart

When the sort is complete and records are ready to be written to the output file, the SortOutStart procedure is called once to initialize the processing of output records. It has the interface:

```
SortOutStart: ErcType
```

This procedure has no parameters. The standard SortOutStart is null. You can substitute a custom version for the standard version by adding initialization logic.

SortOut

The SortOut procedure is called for each output record in turn. The standard SortOut (which is included in SortMerge.Lib) calls OutputRecord (described in section 6) on each record.

To include user-written code for postprocessing output records, you build Sort with your own SortOut procedure that has the interface:

```
SortOut (pRecord, sRecord, iFile): ErcType
```

where

pRecord and sRecord describe the output record to be released.

iFile designates the index of the output file within the specified list of output files (counted from zero for the first file).

The SortOut procedure can modify, delete, or insert output records. You modify output records by passing to OutputRecord a record that is different from the one with which it was called (perhaps reversing a transformation done by a custom SortIn procedure). Delete output records by returning to the calling procedure without calling OutputRecord for selected records. Insert output records by calling OutputRecord more than once on the basis of some computation.

SortOutDone

The SortOutDone procedure is called once after SortOut has been called for all the output records. It has the interface:

```
SortOutDone: ErcType
```

This procedure has no parameters. The standard SortOutDone is null. You can substitute a custom version for the standard version to add termination logic.

Input Error Handling

Whenever Sort detects a malformed input record during the input phase of the sort, it scans forward in the input file for a well-formed record and calls the SortError procedure. (You can replace the standard SortError with a customized version.) The interface is:

```
SortError (iFile,lfaRecord, cbBadData, fConfirm): ErcType
```

where

iFile specifies the number of the input file containing the malformed record (counted from 0).

lfaRecord specifies the 32-bit logical file address of the record within the input file.

cbBadData specifies the number of bytes of data before a well-formed record. A value of 0 means that there are no more records in this input file; a value of -1 means that there are more than 50 sectors of bad data preceding the next well-formed input record.

fConfirm specifies whether you want the opportunity to confirm or deny continuation of the sort operation after Sort detects a malformed input record. Enter FALSE (0) if you specified Yes in the [Suppress Confirmation?] field. Otherwise, fConfirm is TRUE (OFFH).

Prior to calling SortError, Sort displays a status code and writes it to the log file if you specified one.

If SortError returns the status code 0 (Ok), Sort skips the unreadable input records and continues. If SortError returns a status code other than 0, the sort terminates.

If fConfirm is FALSE (0), the standard version of SortError returns a status code of 0. If fConfirm is TRUE (OFFH), the standard version of SortError asks you whether to continue the sort and returns a status code or 0 or nonzero accordingly.

To customize the treatment of errors, you must build the Sort utility with an alternative version of SortError.

Building a Customized Sort Utility

You use the Linker to build a customized Sort utility from the library of Sort object modules, SortMerge.Lib. To activate the Linker, type **Link** in the command field of the Executive command form and press **RETURN**. The following form is displayed:

```
Link
  Object modules          _____
  Run file                _____
  [List file?]            _____
  [Publics?]              _____
  [Line Numbers?]         _____
  [Stack size]            _____
  [Max memory array size] _____
  [Min memory array size] _____
  [System build?]         _____
  [Version]               _____
  [Libraries]             _____
  [DS allocation]         _____
  [Symbol file]           _____
```

Enter [Sys]<Sys>SortMerge.Lib(SortUtility) in the object modules field and Sort.Run in the Run file field. Include in the object modules field any modules containing replacements for the replaceable procedures. Fill in the [Libraries] field with [Sys]<Sys>SortMerge.Lib. Finally, press **GO** to execute the link.

See the *BTOS Linker/Librarian Programming Reference Manual* for more information about the Linker.

Merge Utility

The interactive Merge utility is part of the Sort/Merge package which you activate from the Executive. It merges several preexisting files of sorted data records according to sort keys embedded within those data records.

The files can be any sorted STAM files that you have created with RSAM, DAM, or ISAM. Since the input files must be sorted before they are merged, they usually are the output of either the Sort utility or a prior execution of the Merge utility.

Merge has special features that deal with input files containing malformed records. These are discussed later in this section.

Merge may encounter a record that is out of order in the input. Such a record is called a sequence break. When Merge encounters a sequence break record, Merge writes it to the output file, producing a sequence break in the output. It displays a descriptive status message and writes it to a log file if you have specified one. As with malformed input records, you can customize treatment of sequence breaks during input. Customizing instructions are given later in this section.

In contrast to the other Sort/Merge components (the Sort utility and the Sort object module procedures), Merge does not require temporary disk storage. Because its input is sorted, Merge simply merges all its input files into a merged output file. However, you may need to use temporary files or intermediate Merge operations to merge input files which exceed memory capacity.

While stability is an option in Sort, Merge is always stable; in Merge, two records with equal keys appear in the output in the same order as they appear in the input.

Activating Merge

To activate the Merge utility from the Executive, type **Merge** in the command field of the Executive command form and press **RETURN**. (For further information, see the *BTOS Standard Software Operations Guide*.) The following form displays:

```
Merge
  Input files                 _____________________
  Output file                 _____________________
  Keys                        _____________________
  [Log file]                  _____________________
  [Suppress confirmation?]    _____________________
```

You must fill in the first three fields. The remaining two fields are optional. You can specify the default in an optional field by leaving it blank. After you have filled in the appropriate fields, you press **GO**.

If you wish to check whether or not a single file is sorted, you activate Merge, specify the file as input, and enter [Nul] as output.

Field Descriptions

- Input files specifies a list of the names of one or more sorted files you want to merge. Separate the names with spaces, not commas. Each file must be a STAM file. All valid records in these files are merged; deleted records are skipped. If Merge detects a malformed input record, it activates the error handling facilities described later in this section.
- Output file specifies the name of the file to which you want the output written. The output file is written with RSAM. However, if all of the input records have the same size, the output file is accessible with either DAM or RSAM.
- Keys specifies how sort keys are embedded within each data record. Although the input records can have varying lengths, the records must all have a prefix of common fixed length containing the sort keys.

If you want a multilevel merge, you must enter several specifications in the Keys field. Each specification represents one component of the sort key. Separate the specifications with spaces, not commas. If there is more than one specification, Merge reads the ones that appear first as more significant than the ones that appear later when it determines merge order.

Each key component specification has the form:

TypeName:Length.Offset.AorD.WorM

See the description of these fields in section 3.

- [Log file] specifies the name for the file to which the status report and merge statistics are to be written.

  Merge computes and writes the following statistics to the log file: number of records, number of bytes of data, number of sequence breaks, and elapsed time of merge.

  If you do not specify a log file (the default), no log file is produced. However, all merge statistics and status codes display when the merge is complete.

- [Suppress confirmation?] specifies your desire to monitor error handling.

  If Merge encounters malformed records or sequence breaks in the input file, it displays a descriptive status message and writes it to the log file if you have specified one.

  For malformed records, if you enter **Yes**, Merge automatically skips any malformed input it finds and searches forward in the input data for the next well-formed record.

  If you enter **No**, Merge automatically skips the malformed input and displays a message that asks you whether you want to continue the merge or to terminate.

  For sequence breaks, if you enter **Yes**, Merge displays a message that tells you of the sequence break, and the merge automatically continues. (The message does not require any input from you.)

  If you enter **No**, Merge stops when it encounters a sequence break. Merge displays a message that asks you whether or not you want the merge to proceed.

However, you have an alternative to this method of error handling. Because Merge is supplied not only as a Run file but also as a library of object modules, you can tailor error handling to your requirements by replacing the error handling module. More information on error handling is provided later in this section.

Customizing Merge

The Merge utility is designed to call certain procedures in such a way that the application programmer can customize Merge by replacing these procedures with user-written code.

User-written code is special code that you activate to process all records and provide special sequence break and error handling.

The library of Merge object modules, SortMerge.Lib, includes standard definitions for the following replaceable procedures:

- MergeOutStart
- MergeOut
- MergeOutDone
- MergeSequenceBreak
- MergeError

Merge controls the flow of the merge operation by calling:

1 MergeOutStart once at the beginning of the merge

2 MergeOut for each record in merged order. (For sequence break records, MergeSequenceBreak is called in place of MergeOut. For malformed records, MergeError is called.)

3 MergeOutDone once when Merge is complete

Processing Output Records

MergeOutStart

MergeOutStart is called once at the beginning of the merge. It has the interface:

```
MergeOutStart: ErcType
```

This procedure has no parameters. The standard MergeOutStart is null; it does no work and returns immediately. However, you can substitute a custom version for the standard version to perform initializing or concluding computation.

MergeOut

MergeOut is called for each record in merged order. The standard MergeOut procedure included in SortMerge.Lib calls OutputRecord (whose interface is the same as MergeOut) on its parameter, thus placing the record into the merge output buffer.

To include user-written code for processing output records, you build Merge with your own MergeOut procedure that has the interface:

```
MergeOut (pRecord, sRecord, iFile): ErcType
```

where

pRecord and sRecord describe the input record to be output.

iFile specifies the index of the input file within the designated list of input files (counted from 0).

The MergeOut procedure can modify, delete, or insert output records. You modify output records by passing to OutputRecord a record that is different from the one with which it was called. You can delete output records by returning to the calling procedure without calling OutputRecord for selected records. You can insert output records by calling OutputRecord more than once on the basis of some computation.

Here is an example of a typical custom MergeOut procedure. Suppose the records have fields named Part Number and Quantity Ordered and are merging according to Part Number. A MergeOut Procedure can group sequences of records with the same Part Number and write only a single record for each such group to the output file. The single output record would have the common Part Number and the sum of Quantity Ordered values from the input.

MergeOutDone

MergeOutDone is called once when Merge is complete. It has the interface:

```
MergeOutDone: ErcType
```

This procedure has no parameters. The standard MergeOutDone is null. You can substitute a custom version for the standard version to add termination logic.

Error Handling

Whenever Merge detects a malformed input record during the input phase of the merge, it scans forward in the input file for a well-formed record and calls the MergeError procedure. The interface is:

```
MergeError (iFile, lfaRecord, cbBadData, fConfirm):
ErcType
```

where

iFile specifies the number of the input file containing the malformed record (counting from 0).

lfaRecord specifies the 32-bit logical file address of that record within the input file.

cbBadData specifies the number of bytes of data before a well-formed record. A value of 0 means that there are no more records in this input file; a value of -1 means that there may be up to 50 sectors of bad data preceding the next well-formed input record.

fConfirm specifies whether you want the opportunity to confirm or deny continuation of the merge operation after Merge detects a malformed input record. Enter **FALSE (0)** if you entered **Yes** in the [Suppress confirmation?] field. Otherwise, fConfirm is TRUE (0FFH).

Prior to calling MergeError, Merge displays a status message and writes it to the log file if you specified one.

If MergeError returns the status code 0 (Ok), Merge skips the unreadable input records and continues. If MergeError returns a status code other than 0, the merge terminates.

If fConfirm is FALSE (0), the standard version of MergeError returns a status code of 0. If fConfirm is TRUE (0FFH), the standard version of MergeError asks you whether or not you want to continue the merge and returns 0 or nonzero accordingly.

To customize the treatment of errors, you must build the Merge utility with an alternative version of MergeError.

Sequence Break Handling

Whenever Merge detects a sequence-break record, it calls the MergeSequenceBreak procedure in place of MergeOut. The interface is:

```
MergeSequenceBreak (pRecord, sRecord, iFile, fConfirm):
ErcType
```

where

pRecord and sRecord describe the sequence-break record.

iFile specifies the index, within the specified list of input files, of the input file containing the sequence-break record (counting from 0).

fConfirm specifies whether you want the opportunity to confirm or deny continuation of the merge operation after Merge detects a sequence-break record. If you entered **Yes** in the [Suppress confirmation?] field, specify **FALSE (0)**. Otherwise, fConfirm is TRUE (0FFH).

Prior to calling MergeSequenceBreak, Merge displays a status code and writes it to the log file if you specified one.

If MergeSequenceBreak returns the status code 0 (Ok), the out-of-sequence record is placed in the output and the merge continues. If MergeSequenceBreak returns a status code other than 0, the merge terminates.

The standard version of MergeSequenceBreak returns a status code of 0, if fConfirm is FALSE (0). If fConfirm is TRUE (0FFH), the standard version of MergeSequenceBreak asks you whether or not to continue the merge and returns 0 or nonzero accordingly.

To customize the treatment of sequence breaks, you must build the Merge utility with an alternative version of MergeSequenceBreak.

Building a Customized Merge Utility

You use the Linker to build a customized Merge utility from the library of Merge object modules, SortMerge.Lib. To activate the Linker, you type **Link** in the command field of the Executive command form and press **RETURN**. The following form is displayed:

```
Link
  Object modules      _____________________
  Run file            _____________________
  [List file?]        _____________________
  [Publics?]          _____________________
  [Line numbers?]     _____________________
  [Stack size]        _____________________
  [Max memory array size] _____________________
  [Min memory array size] _____________________
  [System build?]     _____________________
  [Version]           _____________________
  [Libraries]         _____________________
  [DS allocation?]    _____________________
  [Symbol file]       _____________________
```

You enter [Sys]<Sys>SortMerge.Lib(MergeUtility) in the object modules field, and Merge.Run in the run file field. Include in the object modules field any modules containing replacements for the replaceable procedures. You fill in the [Libraries] field with [Sys]<Sys>SortMerge.Lib . You finally, press **GO** to execute the link.

See the *BTOS Linker/Librarian Programming Reference Manual* for more information about the Linker.

Object Module Procedures

Sort/Merge has two types of object module procedures: key-in-record sort procedures and external-key sort procedures. You can link these procedures with an application program and call them from programming languages such as BASIC, FORTRAN, and Pascal. COBOL calls Sort/Merge with the COBOL Sort verb.

Key-In-Record Sort Procedures

In the key-in-record sort object module procedure, records and their associated keys are released to the Sort utility one at a time. When all records are released, the Sort utility does a sort using specified auxiliary disk storage. It then returns the sorted records and associated keys to the application one at a time.

The procedures comprising the key-in-record sort facility are:

- PrepareKeySort
- ReleaseRecord
- DoSort
- ReturnRecord
- ConcludeSort
- TerminateSort

In an application program, you must not mix calls to the key-in-record sort procedures and the external-key sort procedures during the same sort.

Sort controls the flow of the key-in-record sort facility by calling:

1. PrepareKeySort to initialize the Sort/Merge facility. (This specification includes the names of work files and the memory to be used as a sort work area.)
2. ReleaseRecord once for each record to be sorted
3. DoSort to do the actual sort when all records are released
4. ReturnRecord once for each record to retrieve the record and its associated keys in sorted order
5. Conclude Sort to close files and release resources

In the event of an error during the sort, the sort may be prematurely ended and resources released by a call to TerminateSort.

Data Types

The system organizes byte and character data with the most significant byte at the lowest memory address, and binary data with the most significant byte at the highest memory address. Real and packed decimal data are different from each other and from the preceding data, since the sign of the data is stored differently in each case. Therefore, when you use the key-in-record sort, you must properly specify the data types of the sort fields. Once you do this, the extraction of a key and correct comparison of keys is automatic.

PrepareKeySort includes the formula for extracting a sort key from a record. This formula makes possible multilevel sorting by allowing you to specify that a sort key be built by combining several fields of a record.

Each field of a record that comprises its sort key is defined by a key component descriptor whose format is shown in table 5-1.

Table 5-1 **Format of a Key Component Descriptor**

Offset	Field	Size (bytes)	Description
0	rbKey	2	the offset of the key component within the record
2	cbKey	2	the size of the key component in bytes
4	type	2	one of the values 0 to 11 (20 to 31 COBOL), used to represent a key type as described in table 5-2
6	fAscending	2	TRUE (0FFH) if the sorted records are to have ascending values in this field, or FALSE (0H) if they are to have descending values

The fields type and cbKey together specify the type and size of the key component, as shown in table 5-2.

Table 5-2 **Types of Key Components**

Type	Name of Type	Note
0	Binary	cbKey contains the length of the key in bytes. 1 to 8 are valid values.
1	Byte	cbKey contains the length of the key in bytes. 1 to 64 are valid values.
2	Character	cbKey contains the length of the key in bytes. 1 to 64 are valid values.
3	Decimal	cbKey contains (d + 2)/2, where d is (odd) the number of decimal digits in the key. d must not exceed 18.
4	Long Real	cbKey must contain 8.
5	Short Real	cbKey must contain 4.
6	Decimal	See Decimal (odd) for the value of (even) cbKey. This type is used for keys that have an even number of decimal keys.
7	Integer	cbKey contains the length of the key in bytes. 1 to 8 are valid values.
8	Long IEEE	cbKey must contain 8.
9	Short IEEE	cbKey must contain 4.
10	Extended IEEE	cbKey must contain 10.
11	Display	cbKey contains the length of the key in bytes. 1 to 19 are valid values.

Note: *COBOL applications use the values 20 to 31 for the corresponding key types listed in this table.*

Key types and programming language representations are shown in table 5-3.

Table 5-3 **Key Types and Programming Language Representations**

Language and Key Type	Index Spec. cbIndexField	wType
BASIC Interpreter		
Integer (%)	2	7
ShortReal (!)	4	5
Long Real (#)	8	4
BASIC Compiler		
Integer (%)	2	7
ShortReal (!)	4	5
LongReal (#)	8	4
COBOL		
USAGE is DISPLAY (n-byte) (numeric types)	n	31
USAGE is COMP (n-byte) (signed)	n	27
USAGE is COMP (n-byte) (unsigned)	n	20
USAGE is COMP-3 (n-digit) (n even)	(n+2)/2	26
USAGE is COMP-3 (n-digit) (n odd)	(n+1)/2	23
Note: *COBOL uses the types 20 to 31.*		
FORTRAN (Microsoft)		
INTEGER*2	2	7
INTEGER*4	4	7
REAL*4	4	9
REAL*8	8	8
DOUBLE PRECISION	8	8
Pascal (Microsoft)		
Byte	1	0
Integer	2	7
Real	4	9
SInt	1	7
Word	2	0

Key Types

Components of sort keys can have any of these types: binary, byte, character, decimal, long real, short real, integer, IEEE real (short, long, and extended), and display.

Binary

A binary key is a 1- to 8-byte unsigned integer. The high-address byte of a binary key is the most significant for determining sort order. For COBOL COMP fields, the low-address byte is the most significant.

Byte

A byte key is a string of 8-bit bytes. The low-address bytes of the string are the most significant for determining sorting order.

Character

A character key is a string of 8-bit bytes. Character keys are identical to byte keys, except that alphabetic ASCII characters are sorted without regard to their case.

Decimal

A decimal key is a packed decimal number in COBOL COMP-3 format. Each byte contains two decimal digits (four bits per digit) with the digits (0-9) encoded as BCD numbers (0000-1001). The last byte of the key contains the sign and the units digit with the sign in the least significant four bits. The preceding byte contains the tens digit in the least significant four bits, etc.

Decimal fields have the same representation in all programming languages. For more information about this type of field, see your COBOL documentation.

Long/Short Real

Long real and short real keys are used in BASIC applications. A long real key is an 8-byte real number and a short real key is a 4-byte real number.

For information regarding the number of bits of precision and range of values for these keys, see the *BTOS BASIC Compiler Programming Reference Manual*.

Integer

The integer key is a signed 1- to 8-byte integer. The high-address byte of an integer key is the most significant for determining sort order. For COBOL COMP fields, the low-address byte is the most significant.

IEEE Real

Long, short, and extended IEEE keys are used for real numbers in Pascal or FORTRAN applications. The high-address byte is the most significant byte for determining sort order.

Short IEEE Real

The 4-byte IEEE format short real number is used for **REAL*4** in FORTRAN, and for **REAL** in Pascal.

Long IEEE Real

The 8-byte IEEE format long real number is used for **REAL*8** and **DOUBLE PRECISION** in FORTRAN.

Display

A display key is used in COBOL applications for the USAGE IS DISPLAY field. All COBOL sign options are supported. Display keys can be 1 to 19 bytes long. For more information about the range of values and representations for display keys, see your COBOL documentation.

External-Key Sort Procedures

The external-key-sort facility is a component of the Sort/Merge facility that consists of object module procedures. Records and their associated keys are released to the sort package one at a time. When all records are released, the sort package does a sort using specified auxiliary disk storage. It then returns the sorted records and associated keys to the application one at a time. The procedures comprising the external-key sort facility are:

- PrepareSort
- ReleaseRecordAndKey
- DoSort
- ReturnRecordAndKey
- ConcludeSort
- TerminateSort

In an application program, you must not mix calls to the external-key sort procedures and the key-in-record sort procedures during the same sort.

Sort controls the flow of the external-key sort facility by calling:

1. PrepareSort to initialize the Sort/Merge facility. (This specification includes the name of work files and the memory to be used as a sort work area.)
2. ReleaseRecordAndKey once for each record to be sorted
3. DoSort to do the actual sort when all records are released
4. ReturnRecordAndKey once for each record to retrieve the record and its associated keys in sorted order
5. ConcludeSort to close files and release resources

In the event of an error during the sort, the sort may be prematurely ended and resources released by a call to TerminateSort.

Note that the external-key sort procedures interpret the bytes of a key at higher memory addresses as more significant than the bytes at lower memory addresses. In other words, in comparing two keys, the bytes at lower memory addresses are considered only when the bytes at higher memory addresses are equal.

Status Block

Many of the sort procedures take a parameter, which is the memory address of the status block. The sort procedures set this block to report errors to the application program. The format of the 4-byte status block is shown in table 5-4.

Table 5-4 **Status Block Format**

erc	2 bytes	Sort/Merge status code
ercDetail	2 bytes	Detail status code

The status block contains two status codes, erc and ercDetail. The first status code is either 0 (Ok) or one of the Sort/Merge status codes listed in appendix A.

The second status code is nonzero only if erc is nonzero. This status code gives additional information about the error. For example, if a device error occurs while you are trying to open a work file, erc returns the message **Can't open work file** and ercDetail returns the message **I/O error**.

Operations

Sort/Merge has the following nine operations:

- ConcludeSort releases resources after a successful sort.
- DoSort performs the actual sort of released records.
- PrepareKeySort initializes a key-in-record sort.
- PrepareSort initializes an external-key sort.
- ReleaseRecord releases a record for a key-in-record sort.
- ReleaseRecordAndKey releases a record for an external-key sort.
- ReturnRecord returns a sorted record following a key-in-record sort.
- ReturnRecordAndKey returns a sorted record following an external-key sort.
- TerminateSort releases resources following an unsuccessful sort.

Tables 6-1 and 6-2 show the contents of the procedures PrepareSortBlock and KeyDescriptor. Both procedural interfaces are discussed later in this section.

Table 6-1 **Contents of PrepareSortBlock**

Offset	Field	Size (Bytes)	Description
0	filespecWorkfile1	92	the file specification of the first work file. Starting at the second byte of the field, it is a character string of the form [volname]<dirname>filename. The first byte is the length of that string.
92	passwordWorkfile1	13	the file password for the first work file. Starting at the second byte of the array, its length is in the first byte of the array.

Table 6-1 **Contents of PrepareSortBlock (Cont)**

Offset	Field	Size (Bytes)	Description
105	filespecWorkfile2	92	similar to filespecWorkfile1, except that it describes the second work file
197	passwordWorkfile2	13	similar to passwordWorkfile1, except that it describes the second work file
210	qsSortWorkfileCreate	4	the size at which to create the work files
214	sWorkfileIncrement	2	the increment to extend the work files when necessary
216	qsSortWorkArea	4	the size of an already existing work area; or, if 0, it requests that the system allocate all available memory for the work area
220	sRecordMax	2	the maximum size of a record in bytes
222	fStableSort	2	TRUE if a stable sort is desired, and FALSE otherwise. A sort is stable if input records whose sort keys are equal always appear in the output in the same order as they appear in the input.

Table 6-2 **Contents of Key Descriptor**

Offset	Field	Size (Bytes)	Description
0	cKeyComponents	2	the number of key components in each record
2	rgKeyComponent	8	the array of KeyComponent-Descriptor, one entry for each key component (see table 5-1)

ConcludeSort

The ConcludeSort procedure deletes temporary files and closes the work file (deleting them if you created them during PrepareSort) if all the sorted records were retrieved (by ReturnRecord or ReturnRecordAndKey). Otherwise, the status code **More records available** is returned.

The procedural interface is:

ConcludeSort (pStatusBlockRet): ErcType

where

pStatusBlockRet is the memory address of a Status Block (see section 5).

DoSort

The DoSort procedure does the actual sort of all records that were released by ReleaseRecord or ReleaseRecordAndKey.

The procedural interface is:

DoSort (pStatusBlockRet): ErcType

where

pStatusBlockRet is the memory address of a Status Block (see section 5).

PrepareKeySort

The PrepareKeySort procedure initializes the Sort/Merge facility for a key-in-record sort. If more than one key is specified, the earlier keys are more significant than the later ones in determining sort order.

If the two work files specified in the PrepareSortBlock (shown in table 6-1) do not already exist, they are created. Their size is set initially to the value of the field qsSortWorkfileCreate in the PrepareSortBlock. If these work files are created, they are deleted at the end of the sort when ConcludeSort is called (or if TerminateSort is called at any time). If their size is insufficient for the amount of data actually sorted, they are extended as required in specified increments.

A sort work area, which includes the space for file buffers and internal sorting, must be created or specified. If an existing sort work area is used, its address and size have been specified; if the size is specified as 0, PrepareKeySort allocates all unallocated workstation memory for the sort work area.

The procedural interface is:

PrepareKeySort (pPrepareSortBlock, pKeyDescriptor, pSortWorkArea, pStatusBlockRet): ErcType

where

pPrepareSortBlock is the memory address of a PrepareSortBlock (see table 6-1).

pKeyDescriptor is the memory address of a key descriptor (see table 6-2).

pSortWorkArea is the memory address of a work area that may already exist.

pStatusBlockRet is the memory address of a Status Block (see section 5).

This procedure is used by application programs written in BASIC, and FORTRAN. For more information, see appendix B.

PrepareSort

The PrepareSort procedure initializes the Sort-Merge facility for an external-key sort. If the two work files in the PrepareSortBlock (shown in table 6-1) do not already exist, they are created. Their size is set initially to the value of the field qsSortWorkfileCreate in the PrepareSortBlock.

If these work files are created, they are deleted at the end of the sort when ConcludeSort is called (or if TerminateSort is called at any time). If their size is insufficient for the amount of data actually sorted, they are extended as required in specified increments.

A sort work area, which includes the space for file buffers and internal sorting, must be created or specified. If an existing work area is used, its address and size have been specified; if the size is specified as 0, PrepareSort allocates all unallocated workstation memory for the sort work area.

The procedural interface is:

PrepareSort (pPrepareSortBlock, psKey, pSortWorkArea, pStatusBlockRet): ErcType

where

pPrepareSortBlock is the memory address of a PrepareSortBlock (see table 6-1).

psKey is the memory address of a word containing the size of the key in bytes.

pSortWorkArea is the memory address of a work area that may already exist.

pStatusBlockRet is the memory address of a Status Block (see section 5).

This procedure is used by application programs written in BASIC, and FORTRAN. For more information, see appendix B.

ReleaseRecord

The ReleaseRecord procedure releases a record to the Sort facility for a key-in-record sort.

The procedural interface is:

ReleaseRecord (psRecord, pRecord, pStatusBlockRet): ErcType

where

psRecord is the memory address of a word containing the size of the record in bytes. This size must not be greater than the size specified in the call to PrepareKeySort.

pRecord is the memory address of the beginning of the record.

pStatusBlockRet is the memory address of a Status Block (see section 5).

ReleaseRecordAndKey

The ReleaseRecordAndKey procedure releases a record to the Sort facility for an external-key sort.

The procedural interface is:

ReleaseRecordAndKey (psRecord, pRecord, psKey, pKey, PStatusBlockRet): ErcType

where

psRecord is the memory address of a word containing the size of the record in bytes. This size must not be greater than the size specified in the call to PrepareSort.

pRecord is the memory address of the beginning of the record.

psKey is the memory address of a word containing the size of the key in bytes. This size must be the same as the size specified in the call to PrepareSort.

pKey is the memory address of the key.

pStatusBlockRet is the memory address of a Status Block (see section 5).

ReturnRecord

The ReturnRecord procedure returns a sorted record in a key-in-record sort. ReturnRecord should be called repeatedly until it returns the status code **No more records**. The actual freeing of resources and closing of files does not occur until the call to ConcludeSort or TerminateSort.

The procedural interface is:

ReturnRecord (psRecordRet, pRecordRet, pStatusBlockRet): ErcType

where

psRecordRet is the memory address of a word set to the size of the returned record.

pRecordRet is the memory address to which the record is copied. The maximum possible record size is specified at the time of PrepareSortKey.

pStatusBlockRet is the memory address of a Status Block (see section 5).

ReturnRecordandKey

The ReturnRecordAndKey procedure returns a sorted record in an external-key sort. ReturnRecordAndKey should be called repeatedly until it returns the status code **No more records**. The actual freeing of resources and closing of files does not occur until the call to ConcludeSort or TerminateSort.

The procedural interface is:

ReturnRecordAndKey (psRecordRet, pRecordRet, psKeyRet, pKeyRet, pStatusBlockRet): ErcType

where

psRecordRet is the memory address of a word set to the size of the returned record.

pRecordRet is the memory address to which the record is copied. The maximum possible record size is specified at the time of PrepareKeySort.

psKeyRet is the memory address of a word set to the size of the returned key.

pKeyRet is the memory address to which the key is copied. The maximum possible key size is specified at the time of PrepareSort.

pStatusBlockRet is the memory address of a Status Block (see section 5).

TerminateSort

The TerminateSort procedure deletes temporary files and closes (or deletes) the work files. It should be called if the sort is to be terminated (for example, if an error is detected) prior to the time when all records are retrieved.

The procedural interface is:

TerminateSort (pStatusBlockRet): ErcType

where

pStatusBlockRet is the memory address of a Status Block (see section 5).

Status Codes

General

Decimal Value	Meaning
3200	Invalid key type. The type field of a key specification for Sort/Merge is invalid.
3201	Incorrect key length. The cbKey field of a key specification for a Sort/Merge operation does not correspond to the type field of the key specification. (For example, for binary keys, cbKey must be 2.)
3202	Invalid key. A key contained in a record for Sort/Merge is not of the correct type. (For example, each digit of a BCD key must be between 0 and 9.)

External-Key Sort

Decimal Value	Meaning
3400	Cannot open work file. Unable to open one of the work files during PrepareSort.
3401	Work area invalid. Unable to allocate work area during PrepareSort.
3402	Invalid key size. A key passed to ReleaseRecordAndKey is a different length from the length specified in PrepareSort.
3403	File error during sort. A file error occurred during the sort phase of the program.
3404	No more records. ReturnRecordAndKey was called after all records were retrieved.
3405	Error returning record. An error occurred in ReturnRecordAndKey.

External-Key Sort (Cont)

Decimal Value	Meaning
3406	Error during conclude. An error occurred in ConcludeSort or TerminateSort.
3407	More records available. ConcludeSort was called before all records were retrieved. To end a sort prematurely, call TerminateSort.
3408	Record too large. The size of a record is larger than the maximum key size specified in PrepareSort, or the sort area is not large enough.
3409	Error during sort. An error occurred during DoSort.
3410	Insufficient memory. Not enough memory was allocated for the sort work area.
3411	No records to sort. DoSort was called before any records were released.
3412-3499	Reserved.

Key-In-Record Sort

Decimal Value	Meaning
3500	Sort pending. PrepareKeySort was called while a sort was already active.
3501	No sort pending. A sort procedure other than PrepareKeySort was called before PrepareKeySort.
3502	Invalid sort key. The key provided is inconsistent with its specifications.

Key-In-Record Sort (Cont)

Decimal Value	Meaning
3503	Sort key not in record. A key could not be synthesized from this record, given the initial specifications of keys within records.
3504	Invalid key specification. The key specification in PrepareKeySort is incorrect. It conflicts with the maximum record size provided.
3505-3529	Reserved.

Sort Utility

Decimal Value	Meaning
3530	Invalid key specification. The key specification passed to Sort is invalid.
3531	Non-numeric key length. The length field of the key specification is non-numeric.
3532	Record too large. A record found in the file to be sorted is too large.
3533	Malformed record. A record found in the file to be sorted is malformed.
3534-3559	Reserved.

Merge Utility

Decimal Value	Meaning
3560	Invalid key specification.
	The key specification passed to Merge is invalid.
3561	Non-numeric key length.
	The length field of the key specification is non-numeric.
3562	Record too large.
	A record found in a file to be merged is too large.
3563	Insufficient memory.
	There is not enough memory available to perform this merge.
3564	Sequence break.
	A sequence break has occurred in one or more of the files being merged. A sort of that file must be performed first.
3565	Malformed record.
	A record found in a file to be merged is malformed.
3566-3599	Reserved.

Calling Sort Object Modules From Programming Languages

Use the procedures BasicPrepareKeySort and BasicPrepareSort (described in this appendix) in place of PrepareKeySort and PrepareSort to call Sort object modules from BASIC and FORTRAN. The former procedural interfaces give easier access to Sort object modules.

COBOL calls Sort by using the COBOL SORT verb. (For more information, see your COBOL documentation.)

BasicPrepareKeySort

The BasicPrepareKeySort procedure has the same effect as PrepareKeySort, but provides a more useful interface to BASIC and other languages.

The BasicPrepareKeySort procedure initializes the Sort/Merge facility for a key-in-record sort.

If the two work files specified in the PrepareSortBlock (shown in table 6-1) do not already exist, they are created. Their size is set initially to the value of the field qsSortWorkfileCreate in the PrepareSortBlock. If these files are created, they are deleted at the end of the sort when ConcludeSort is called (if TerminateSort is called at any time).

If the size of the work files is insufficient for the amount of data actually sorted, it is extended as required in specified increments. A sort work area, which includes the space for file buffers and internal sorting, must be created or specified. If an existing sort work area is used, its address and size have already been specified; if the size is specified as 0, BasicPrepareKeySort allocates all unallocated workstation memory for the sort work area.

Procedural Interface

BasicPrepareKeySort (pPrepareSortBlock,
pbFileSpecWorkfile1, bFileSpecWorkfile1,
pbPasswordWorkfile1, cbPasswordWorkfile1,
pbFileSpecWorkfile2, cbFileSpecWorkfile2,
pbPasswordWorkfile2, cbPasswordWorkfile2,
qsWorkfileCreate, sWorkfileIncrement, qsSortWorkArea,
sRecordMax, fStableSort, pKeyDescriptor, pSortWorkArea,
pStatusBlockRet): ErcType

where

pPrepareSortBlock is the memory address of a space allocated for the PrepareSortBlock shown in table 6-1. PrepareSortBlock is filled in by the BasicPrepareKeySort procedure from the other parameters. The allocated space must be at least 224 bytes.

pbFileSpecWorkfile1 andcbFileSpecWorkfile1 describe the file specification of the first work file.

pbPasswordWorkfile1 and cbPasswordWorkfile1 describe the file password for the first work file.

pbFileSpecWorkfile2 and cbFileSpecWorkfile2 describe the file specification of the second work file.

pbPasswordWorkfile2 and cbPasswordWorkfile2 describe the file password for the second work file.

qsWorkfileCreate is the size at which to create the work files.

sWorkfileIncrement is the increment to extend the work files when they need to be extended.

qsSortWorkArea is the size of an already existing work area or, if 0, it requests that the system allocate all free memory for the work area.

sRecordMax is the maximum size of a record in bytes.

fStableSort is TRUE (0FFH) if a stable sort is desired, and FALSE (0H) otherwise. A sort is stable if input records whose sort keys are equal always appear in the output in the same order as they appear in the input.

pKeyDescriptor is the memory address of a key descriptor (see table 6-2).

pSortWorkArea is the memory address of a work area that may already exist (ignored if qsSortWorkArea equals zero).

pStatusBlockRet is the memory address of the status block into which the status codes from the operation are returned (see section 5).

BasicPrepareSort

The BasicPrepareSort procedure initializes the Sort/Merge facility for an external-key sort.

The procedure has the same effect as PrepareSort, but provides a more useful interface to BASIC and other languages.

If the two work files specified in the PrepareSortBlock (shown in table 6-1) do not already exist, they are created. Their size is set initially to the value of the field qsSortWorkfileCreate in the PrepareSortBlock. If these work files are created, they are deleted at the end of the sort when ConcludeSort is called (or if TerminateSort is called at any time).

If the work area size is insufficient for the amount of data actually worked, it is extended as required in specified increments. A sort work area, which includes the space for file buffers and internal sorting, must be created or specified. If an existing sort work area is used, its address and size have already been specified; if the size is specified as 0, BasicPrepareSort allocates all unallocated workstation memory for the sort work area.

Procedural Interface

BasicPrepareKeySort (pPrepareSortBlock,
pbFileSpecWorkfile1, cbFileSpecWorkfile1,
pbPasswordWorkfile1, cbPasswordWorkfile1,
pbFileSpecWorkfile2, cbFileSpecWorkfile2,
pbPasswordWorkfile2, cbPasswordWorkfile2,
qsWorkfileCreate, sWorkfileIncrement, qsSortWorkArea,
sRecordMax, psKey, pSortWorkArea, pStatusBlockRet):
ErcType

where

pPrepareSortBlock is the memory address of a space allocated for the PrepareSortBlock shown in table 6-1. PrepareSortBlock is filled in by the BasicPrepareKeySort procedure from the other parameters. The allocated space must be at least 224 bytes.

pbFileSpecWorkfile1 and cbFileSpecWorkfile1 describe the file specification of the first work file.

pbPasswordWorkfile1 andcbPasswordWorkfile1 describe the file password for the first work file.

pbFileSpecWorkfile2 and cbFileSpecWorkfile2 describe the file specification of the second work file.

pbPasswordWorkfile2 and cbPasswordWorkfile2 describe the file password for the second work file.

qsWorkfileCreate is the size at which to create the work files.

sWorkfileIncrement is the increment to extend the work files when they need to be extended.

qsSortWorkArea is the size of an already existing work area or, if 0, it requests that the system allocate all free memory for the work area.

sRecordMax is the maximum size of a record in bytes.

psKey is the memory address of a word containing the size of the key in bytes.

pSortWorkArea is the memory address of a work area that may already exist (ignored if qsSortWorkArea equals zero).

pStatusBlockRet is the memory address of the status block into which the status codes from the operation are returned (see section 5).

Glossary

Absolute symbol An absolute symbol is a symbol that has a specified place in memory (as, for example, an address within BTOS).

Address expression An address expression is a description consisting of one or more symbols, or an indexed or nonindexed parameter.

Alignment attribute An alignment attribute specifies whether the segment can be aligned on a byte, word, or paragraph boundary.

Application partition An application partition is a section of user memory reserved for the execution of an application.

Applications Applications are programs that provide a complete user interface.

ASCII ASCII, the American Standard Code for Information Interchange, defines the character set codes used for information exchange between equipment.

Assemble **ASSEMBLE** is the Executive command you use to display the Assembler command form.

Assembler The Assembler translates Assembly 8086 programs into BTOS object modules (machine code).

Asynchronous Terminal Emulator The Asynchronous Terminal Emulator (ATE) allows a workstation to emulate an asynchronous character-oriented ASCII terminal (glass TTY).

ATE See Asynchronous Terminal Emulator.

BASIC BASIC is one of the high level languages you can use to write BTOS programs. You can use the BASIC Compiler to convert the programs into BTOS object modules, or you can use the BASIC Interpreter to edit and run BASIC programs.

BSWA See Byte Stream Work Area.

Byte stream A byte stream (part of the Sequential Access Method) is a readable or writable sequence of 8-bit bytes.

Byte stream work area The Byte Stream Work Area (BSWA) is a 130-byte memory work area for the exclusive use of SAM procedures.

Class name A class name is a symbol used to designate a class.

Client process A client process requests system service. Any process can be a client process, since any process can request system service.

COBOL COBOL is one of the high level languages you can use to write BTOS programs. You can use the COBOL Compiler to convert the programs into BTOS object modules.

Code listing A code listing is an English-language display of compiled code.

Code segment A code segment is a variable-length (up to 64KB) logical entity consisting of reentrant code and containing one or more complete procedures.

Compiler BTOS Compilers translate high level language programs into BTOS object modules (machine code).

Configuration file Configuration files specify the characteristics of the parallel printer, serial printer, or other devices attached to a communications channel.

Crash dump A crash dump is the output (memory dump) resulting from a system failure.

CTOS.lib The CTOS.lib file is part of the Language Development software; it is a library of object modules that provide operating system run time support.

Cursor RAM The cursor RAM allows software to specify a 10-bit by 15-bit array as a pattern of pixels in place of the standard cursor.

Customizer The BTOS Customizer software provides object module files that allow you to customize the operating system.

DAM See Direct Access Method.

DAWA See Direct Access Work Area.

DCB See Device Control Block.

Device control block A memory-resident Device Control Block (DCB) exists for each device. The DCB contains device information generated at system build. (For a disk, the information includes the number of tracks and sectors per track.)

DGroup DGroup usually includes data, constant, and stack Linker segments.

Direct access method The Direct Access Method (DAM) provides random access to disk file records identified by record number. When you create the DAM file, you specify the record size.

DAM supports COBOL Relative I/O and any BTOS language program can use a direct call for DAM.

Direct access work area A Direct Access Work Area (DAWA) is a 64-byte memory work area for the exclusive use of the Direct Access Method (DAM) procedures.

$ Directories When BTOS receives a request with the directory $, the directory name is expanded to $nnn. (nnn represents the application user number.)

Double-precision Double-precision parameters designate two words to store an item of data to maintain a high level of precision.

DS allocation An option in the Linker, DS allocation locates DGroup at the end of a 64KB segment that the DS register addresses.

8086 Assembly Language 8086 Assembly language is the low level language you can use to write BTOS programs. You use the BTOS Assembler to convert the programs into BTOS object modules.

Environment An environment is a program that has control of the system at any given time. Environments include the SignOn form, the Executive, the Mail Manager, utilities (such as Floppy Copy), applications (such as a word processor), and Compilers.

Escape sequence An escape sequence is a sequence of characters that activates a function.

Executive The Executive is the BTOS user interface program; it provides access to many convenient utilities for file management.

External reference An external reference is a reference from one object module to variables and entry points of other object modules.

Extraction Librarian extraction copies an object module from a library into a separate disk file. Extraction does not delete the extracted module from the library.

Field A field is an area in a display form that contains parameters.

File access methods Several file access methods augment the file management system capabilities. File access methods are object module procedures located in the standard BTOS library. They provide buffering and use the asynchronous input/output capabilities of the file management system to overlap input/output and computation.

Font The BTOS Font Designer software allows programmers to design or edit characters by drawing or erasing pixels.

Forms The BTOS Forms software allows programmers to design user-entry forms for applications.

Forms.lib The Forms.lib file is part of the Language Development software; it is an object module library for Forms Run Time support.

FORTRAN FORTRAN is one of the high level languages you can use to write BTOS programs. You can use the FORTRAN Compiler to convert the programs into BTOS object modules.

Group A group is a named collection of linker segments that the BTOS loader addresses at run time with a common hardware segment register. To make the addressing work, all the bytes within a group must be within 64K of each other.

Indexed address An indexed address is an address expression that uses index registers.

Indexed Sequential Access Method The BTOS Indexed Sequential Access Method (ISAM) provides random access to fixed-length records identified by multiple keys stored in disk files.

ISAM See Indexed Sequential Access Method.

Language Development The BTOS Language Development software provides the Linker, Librarian, and Assembler programs (**LINK**, **LIBRARIAN**, and **ASSEMBLE** Executive commands).

LED LED stands for light-emitting diode (the red light on a keyboard key).

.lib .lib is the standard file name suffix for library files.

Librarian The Librarian is a program that creates and maintains object module libraries. The Linker can search automatically in such libraries to select only those object modules that a program calls.

Library A library is a stored collection of object modules (complete routines or subroutines) that are available for linking into run files.

Library file A library file can contain one or more object modules. The file name normally includes the suffix .lib.

Link **LINK** is the Executive command that displays the Linker command form.

Linked-list data structure A linked-list data structure contains elements that link words or link pointers connect.

Linker The Linker is a program that combines object modules (files that Compilers and Assemblers produce) into run files.

Linker segment A Linker segment is a single entity consisting of all segment elements with the same segment name.

Link pointer A link pointer is a 32 bit address that points to the next block of data.

Link word A link word is a 16 bit address that points to the next block of data.

List file The Linker list file (suffix .map) contains an entry for each Linker segment, identifying the segment relative address and length in the memory image. You can direct the Linker to list public symbols and line numbers.

Long-lived memory Long-lived memory is an area of memory in an application partition. It is used for parameters or data passed from an application to a succeeding application in the same partition.

.map .map is the standard file name suffix for list files.

Memory array A memory array is data space the BTOS Loader allocates above the highest task address.

.obj .obj is the standard file name suffix for object module files.

Object module An object module is the result of a single Compiler or Assembler function. You can link the object module with other object modules into BTOS run files.

Offset The offset is the number of bytes between the beginning of a segment and the memory location.

Overlay An overlay is a code segment made up of the code from one or more object modules. An overlay is loaded into memory as a unit and is not permanently memory-resident. See also virtual code segment management.

Parameter A parameter is a variable or constant that is transferred to and from a subroutine or program.

Pascal Pascal is one of the high level languages you can use to write BTOS programs. You can use the Pascal Compiler to convert the programs into BTOS object modules.

Physical address A physical address is an address that does not specify a segment base and is relative to memory location 0.

Pixels Pixels are square-shaped cells which make up the dot matrix of a character symbol.

Pointer A pointer is an address that specifies a storage location for data.

Process A process is a program that is running.

Public procedure A public procedure is a procedure that has a public address; a module other than the defining module can reference the address.

Public symbol A public symbol is an ASCII character string associated with a public variable, a public value, or a public procedure.

Public value A public value is a value that has a public address; a module other than the defining module can reference the address.

Public variable A public variable is a variable that has a public address; a module other than the defining module can reference the address.

Record sequential access method Record Sequential Access Method (RSAM) files are sequences of fixed-length or variable-length records. You can open the files for read, write, or append operations.

Relocation The BTOS Loader relocates a task image in available memory by supplying physical addresses for the logical addresses in the run file.

Relocation directory The relocation directory is an array of locators that the BTOS Loader uses to relocate the task image.

Resident The resident portion of a program remains in memory throughout execution.

Reverse video Reverse video displays dark characters on a light screen.

RSAM See Record Sequential Access Method.

.run .run is the standard file name suffix for run files.

Run file A run file is a complete program: a memory image of a task in relocatable form, linked into the standard format BTOS requires. You use the Linker to create run files.

Run file checksum The Run-file checksum is a number the Linker produces based on the summation of words in the file. The system uses the checksum to check the validity of the run file.

SAM See Sequential Access Method.

SamGen See SAM Generation.

SAM Generation SAM generation permits the specification of device-dependent object modules to be linked to an applica?tion.

Segment A segment is a contiguous area of memory that consists of an integral number of paragraphs. Segments are usually classified into one of three types: code, static data, or dynamic data. Each kind can be either shared or nonshared.

Segment address The segment address is the segment base address. For an 8086/80186 microprocessor, a segment address refers to a paragraph (16 bytes).

Segmented address A segmented address is an address that specifies both a segment base and an offset.

Segment element A segment element is a section of an object module. Each segment element has a segment name.

Segment override Segment override is operating code that causes the 8086/80186 to use the segment register specified by the prefix instead of the segment register that it would normally use when executing an instruction.

Sequential access method Sequential Access Method (SAM) files emulate a conceptual, sequential character-oriented device known as a byte stream to provide device-independent access to devices.

Short-lived memory Short-lived memory is the memory area in an application partition. When BTOS loads a task, it allocates short-lived memory to contain the task code and data. A client process can also load short-lived memory in its own partition.

Stack A stack is a region of memory accessible from one end by means of a stack pointer.

Stack frame The stack frame is a region of a stack corresponding to the dynamic invocation of a procedure. It consists of procedural parameters, a return address, a saved-frame pointer, and local variables.

Stack pointer A stack pointer is the indicator to the top of a stack. The stack pointer is stored in the registers SS:SP.

Submit file escape sequence A submit file escape sequence consists of two or three characters that indicate the presence of the escape sequence (% or >), followed by a code to identify the special function, followed by an argument to the function.

.sym .sym is the standard file name suffix for the symbol file.

Symbol Symbols can be alphanumeric and/or any other characters, such as underscore, period, dollar sign, pound sign, or exclamation mark.

Symbol file The Linker symbol file (suffix .sym) contains a list of all public symbols.

Symbolic instructions Symbolic instructions are instructions containing mnenomic characters corresponding to Assembly language instructions. These instructions cannot contain user-defined public symbols.

Sys.Cmds The Executive command file ([Sys]<sys>Sys.Cmds) contains information on each Executive command.

System build System build is the collective name for the sequence of actions necessary to construct a customized BTOS image.

System image The system image file ([Sys]<sys>SysImage.Sys) contains a run file copy of BTOS.

System partition The system partition contains BTOS and dynamically installed system services.

System process A system process is any process that is not terminated when the user calls Exit.

System service process A system service process is an operating system process that services and responds to requests from client processes.

Task A task consists of executable code, data, and one or more processes.

Task image A task image is a program stored in a run file that contains code segments and/or static data segments.

Text file A text file contains bytes that represent printable characters or control characters (such as tab, new line, etc.).

UCB See User Control Block.

Unresolved external reference An unresolved external reference is a public symbol that is not defined, but is used by the modules you are linking.

User control block The User Control Block (UCB) contains the default volume, directory, password, and file prefix set by the last Set Path or Set Prefix operation.

User process A user process is any process that is terminated when the user calls Exit.

Utilities Utilities are programs that use the Executive user interface (such as Floppy Copy or Ivolume).

Video attributes Video attributes control the presentation of characters on the display.

Virtual code segment management Virtual code segment management is the virtual memory method BTOS supports. The method works as follows: The Linker divides the code into task segments that reside on disk (in the run file). As the run file executes, only the task segments that are required at a particular time reside in the application partition's main memory; the other task segments remain on disk until the application requires them. When the application no longer requires a task segment, another task segment overlays it.

Index

Title: BTOS Sort/Merge Operations Reference Manual

Form Number: 5022148 Date: January 1986

Burroughs Corporation is interested in your comments and suggestions regarding this manual. We will use them to improve the quality of your Product Information.

Please check type of suggestion: ☐ Addition ☐ Deletion ☐ Revision ☐ Error

Comments: _____

_____

_____

_____

Name _____

Title _____

Company _____

Address _____
Street City State Zip

Telephone Number ( ) _____
Area Code

Title: BTOS Sort/Merge Operations Reference Manual

Form Number: 5022148 Date: January 1986

Burroughs Corporation is interested in your comments and suggestions regarding this manual. We will use them to improve the quality of your Product Information.

Please check type of suggestion: ☐ Addition ☐ Deletion ☐ Revision ☐ Error

Comments: _____

_____

_____

_____

Name _____

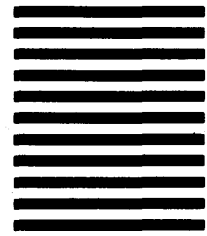
Title _____

Company _____

Address _____
Street City State Zip

Telephone Number ( ) _____
Area Code

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES



BUSINESS REPLY CARD

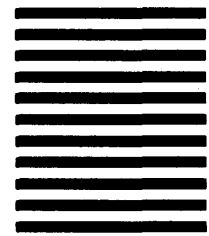
FIRST CLASS PERMIT NO. 817 DETROIT, MI 48232

POSTAGE WILL BE PAID BY ADDRESSEE

Burroughs Corporation
1300 John Reed Court
City of Industry, CA 91745 USA

ATTN: Corporate Product Information

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES



BUSINESS REPLY CARD

FIRST CLASS PERMIT NO. 817 DETROIT, MI 48232

POSTAGE WILL BE PAID BY ADDRESSEE

Burroughs Corporation
1300 John Reed Court
City of Industry, CA 91745 USA

ATTN: Corporate Product Information

Burroughs

Printed in U.S.