999-801-315IS

AT&T UNIX® PC Sort/Merge Programmer's Guide

PREFACE

This manual describes the AT&T **UNIX**™ PC Sort/Merge package, which sorts records by one or more key fields and merges the sorted records. Sort/Merge can be called directly or indirectly from an application program written in AT&T **UNIX** PC BASIC, C, SVS-FORTRAN™ , or SVS-Pascal™ . This manual is oriented toward the experienced UNIX application programmer.

Section 1, Overview, presents Sort/Merge features and procedures.

Section 2, Procedures, describes the format of the Sort/Merge procedures in detail.

Additional information is provided in the following appendices:

Appendix A lists error and exception codes.

Appendix B presents the internal definition for Sort/Merge data structures.

Appendix C explains how to manually remove the temporary work files used by Sort/Merge when Sort/Merge terminates abnormally.

Appendix D provides examples of programs written in BASIC, C, FORTRAN, and Pascal that access Sort/Merge.

Installing **UNIX** PC Sort/Merge From the User Agent

To install Sort/Merge on your **UNIX** PC, see Installing Optional **Application Software** in the AT&T **UNIX**™ PC Installation Guide.

Note

The Development Set, included with the Optional UNIX Utilities set, must be installed in order to use AT&T **UNIX** PC Sort/Merge.

CONTENTS

Contents

TABLES

SECTION 1: OVERVIEW

UNIX PC Sort/Merge sequences records by comparing one or more key fields (keys) contained in the records. After comparing the keys, Sort/Merge sorts the records. Calls to Sort/Merge are made directly or indirectly from any of the AT&T **UNIX** PC programming languages--BASIC, C, SVS-FORTRAN™ , or SVS-Pascal™ .

FEATURES

UNIX PC Sort/Merge does the following:

- Sorts variable-length records on fixed-length keys.
- Sorts records of several different data types (binary, byte, character, LongReal, or ShortReal).
- Sorts records in ascending order (lower values to higher values).
- Sorts records in descending order (higher values to lower values).
- Sorts records using a single key (a single-level sort).
- Sorts records using more than one key (a multilevel sort).
- Sorts and merges records from more than one source.
- Responds to calls from any supported application program.
- Internationalizes the collating sequence used for comparing keys.
- Sorts records up to 16000 bytes in length.

CONCEPTS

Sort/Merge sequences records by comparing values contained in key fields (keys) in the records and then sorting the records. For

example, the following records are sorted by part name in ascending order:

Part Name	Backlog No.
Alternator	100
Carburetor	200
Manifold	200
Radiator	100
Regulator	200

Since the records are sorted by part name, part name is the key field (key). If the records were sequenced by backlog number, backlog number would be the key. The example above is a single-level sort, because only one field in the record is used as a key. But as many keys as there are fields in the record can be specified for sequencing. For example, the records shown above can be sorted first by backlog number in descending order and, for records with the same backlog number, by part name in ascending order. The result looks like

Part Name	Backlog No.
Carburetor	200
Manifold	200
Regulator	200
Alternator	100
Radiator	100

This is an example of a multilevel sort. The combination of more than one sort key for sequencing is called a composite sort key.

The application programmer designates the keys and other values used for comparing and sequencing records. For each key specified, the following four values must be defined:

1. Position of the field in the record (offset from the first byte of the record).
2. Length of the field in bytes.
3. Data type (binary, byte, character, longreal, or shortreal).
4. Sort order (either ascending or descending).

The set of values associated with each key is a sort key.

PROCEDURES

UNIX PC Sort/Merge comprises the six procedures described briefly below. The procedures are listed in the order in which they are most likely to be called within a program. The format of each procedure is detailed in Section 2, "Procedures."

- PrepareKeySort initiates Sort/Merge, allocates work space, and creates temporary work files. PrepareKeySort also describes the keys used to sort records. The PrepareKeySort parameters specify the number of key fields in a sort key, the position and length of each field, the data type (e.g., real), and the sort order (either ascending or descending).

- ReleaseRecord passes records one at a time from the calling program to the sort procedure until all records have been released. The ReleaseRecord parameters specify the memory address and record size.

- DoSort performs the sort function after all records have been released by ReleaseRecord.

- ReturnRecord returns records one at a time in sorted order to the application program work area. ReturnRecord should be called as many times as necessary to return all records. The ReturnRecord parameters specify the address of the buffer where the record should be placed and the address where the record's size should be placed.

- ConcludeSort releases the work space and removes the temporary work files used by Sort/Merge, if all sorted records have been returned to the application program and the sort has been successfully completed.

- TerminateSort stops a sort before it is completed, releases the work space, and removes the temporary files used by Sort/Merge. TerminateSort should be called whenever an error code is returned by any of the sort procedures.

SECTION 2: PROCEDURES

This section describes the format of each **UNIX** PC Sort/Merge procedure. The procedures are arranged in alphabetical order. The conventions used to place calls and to name parameters vary from one program language to another.

PROCEDURE STATUS

Each procedure returns a status code (ErcType) that reports the status of the procedure to the calling program. A zero (0) indicates successful completion of the procedure; a negative integer indicates an error; and a positive integer indicates an exception. (Refer to Appendix A, Status Codes, for a detailed description of error codes.)

NOTATION FOR PARAMETER NAMES

Many of the parameter names used in this section include prefixes that by convention denote parameter information. The prefixes are listed below.

- c = count. For example, cSortKeys specifies the number of sort keys in a record.
- f = flag (TRUE = not 0, FALSE = 0). For example, if fAscending is set to 1, the field is sorted in ascending order.
- p = logical memory address (pointer). For example, pKeyDescriptor specifies the address of a KeyDescriptor.
- rg = array of. For example, rgKeyComponentDescriptors is the array of KeyDescriptors.
- s = size, in bytes (unsigned number). For example, sRecord specifies the number of bytes in a record.

NAME

ConcludeSort

SYNOPSIS

```
ConcludeSort()
```

DESCRIPTION

ConcludeSort releases the work space and removes the temporary work files used by Sort/Merge if ReturnRecord successfully returns all sorted records to the calling program. If all records have not been returned to the calling program, a status code 2 (More records available) is returned, and no action is taken by ConcludeSort.

NAME
DoSort

SYNOPSIS
DoSort()

DESCRIPTION
DoSort performs the actual sort of all records.

NAME

PrepareKeySort

SYNOPSIS

```
int PrepareKeySort(pPrepareSortBlock,pKeyDescriptor)
struct {
       long MaxRecordLength;
       long fStable;
}      *pPrepareSortBlock;

struct {
       long cSortKeys;
       struct {
            long KeyOffset;
            long KeyLength;
            long KeyType;
            long fAscending;
       } sortkeys[# of keys];
} *pKeyDescriptor
```

DESCRIPTION

PrepareKeySort initializes Sort/Merge, allocates work space, and creates temporary work files. PrepareKeySort parameters specify the sort keys. The PrepareKeySort procedure must be called before any of the other Sort/Merge procedures.

A detailed interpretation of the arguments in the call is:

pPrepareSortBlock is the logical memory address (pointer) of a PrepareSortBlock. (The content of PrepareSortBlock is detailed in Table 2-1 below.)

pKeyDescriptor is the logical memory address (pointer) of a KeyDescriptor structure, which specifies all sort keys. (The content of KeyDescriptor is detailed in Table 2-2 below.)

TABLE 2-1. PrepareSortBlock

Offset	Field	Size of Field (in bytes)	Parameter
0	MaxRecord Length	4	Maximum size of a record in bytes. This value cannot exceed 16000.
4	fStable	4	TRUE(not 0) = stable sort; FALSE(0) = not stable. (A stable sort maintains the order of records with equal keys.)

TABLE 2-2. KeyDescriptor

Offset	Field	Size of Field (in bytes)	Parameter
0	cSortKeys	4	Number of keys in a record (same as number of sort levels).
4	rgKeyComponent-Descriptor	16*cSortKeys	Array of KeyDescriptors (keys) that forms the sort key. For each key (sort level) in the record, specify an entry in this array, consisting of four Key Component Descriptors. (The content of each Key Component Descriptor is detailed in Table 2-3 below.)

TABLE 2-3. KeyComponentDescriptor(s)

Offset	Field	Size of Field (in bytes)	Parameter
0	KeyOffset	4	Offset of the key field in the record.
4	KeyLength	4	Size of the key component in bytes. KeyLength and KeyType must agree. (The values for KeyLength are presented in Table 2-4 below.)
8	KeyType	4	One of six data types. (The values for KeyType are presented in Table 2-4 below.)
12	fAscending	4	Sort order. TRUE(not 0) = ascending; FALSE(0) = descending.

TABLE 2-4. Key Data Types

KeyType	Type Name/Definition	KeyLength
0	Binary (32-bit unsigned integer)	4
1	Byte (8-bit character strings. Low address is the most significant for determining sort order.)	Number of bytes in the key.
2	Character (8-bit character strings. Identical to byte keys, except alphabetic ASCII characters are sorted without regard to typographic case.)	Number of bytes in the key.
4	LongReal (64-bit valid real number used in BASIC, C, FORTRAN, and Pascal.)	8
5	ShortReal (32-bit valid real number used in BASIC, C, FORTRAN, and Pascal.)	4

NAME

ReleaseRecord

SYNOPSIS

```
int ReleaseRecord(sRecord,pRecord)
long sRecord;
char *pRecord;
```

DESCRIPTION

ReleaseRecord passes records, one at a time, from the calling program until all records have been released.

A detailed interpretation of the arguments in the call is:

sRecord is the size, in bytes, of the record being released. (Record size cannot exceed the size specified in the call to PrepareKeySort.)

pRecord is the memory address of the record being released.

NAME

ReturnRecord

SYNOPSIS

```
int ReturnRecord(psRecordRet,pRecordRet)
short int *psRecordRet;
char *pRecordRet;
```

DESCRIPTION

ReturnRecord returns sorted records, one at a time, to the calling program until all records have been returned and a status code 3 (No more records) is returned.

A detailed interpretation of the arguments in the call is:

psRecordRet is the address in the calling program, which will receive the size of the record. (The record size cannot exceed that specified in PrepareKeySort.)

pRecordRet is the address that will receive the record.

NAME

TerminateSort

SYNOPSIS

TerminateSort()

DESCRIPTION

TerminateSort stops an unfinished sort, releases work space, and removes temporary work files. This procedure can be used to terminate a sort before all records have been retrieved or when an error code is detected.

If Sort/Merge terminates abnormally and TerminateSort does not run, remove the work files manually. (Refer to Appendix C, Removing Sort/Merge Temporary Work Files Manually, for detailed instructions.)

TerminateSort should be called whenever an error code (non-zero) is returned from any Sort/Merge routine.

APPENDIX A: STATUS CODES

Each procedure returns a status code (ErcType) that reports the status of the procedure to the calling program. A zero (0) indicates successful completion of the procedure; a negative integer indicates an error; and a positive integer indicates an exception.

ERROR CODES

Decimal Value	Description
-1	Bad record length. Record length passed to ReleaseRecord exceeds maximum length specified in PrepareKeySort.
-7	File Create failed. File system could not create a temporary work file.
-8	Bad key specification. KeyType and KeyLength specifications not in agreement. For example, if a binary KeyType is specified, the KeyLength must equal 4. This error is also reported on MaxRecordLength < = 0, cSortKeys < = 0, KeyLength < = 0.
-9	Record area too small. Record size passed to ReleaseRecord is smaller than the size to accommodate all keys. Record returned by ReturnRecord is larger than the maximum specified in PrepareKeySort.

Status Codes

Decimal Value	Description
-11	SortKey not in record. Key specifications do not match the requirements of the record.
-16	Bad work area. Cannot allocate work space.
-18	Cannot read from input file (including work file).
-22	Cannot write to output file (including work file).
-24	No sort pending. PrepareKeySort called after another sort procedure called.
-25	Record area too large. The record passed to ReleaseRecord is larger than MaxRecordLength.
-26	Sequence break. A Sort/Merge procedure has been called out of sequence.
-28	Bad BCD key.
-29	ercNoDiskSpace.

EXCEPTION CODES

Decimal Value	Description
2	More records available. Not all sorted records returned to file; cannot delete files or deallocate work space; call ReturnRecord until procedure completed.

Decimal Value	Description
3	No more records. All sorted records returned to file; call ConcludeSort.

APPENDIX B: INTERNAL DEFINITION FOR DATA STRUCTURES

This section gives the internal definition for the AT&T **UNIX** PC Sort/Merge data structures.

```
/*    Internal definition for sort/merge data structures  */

#include <stdio.h>

typedef
        struct {
             int MaxRecordLength;
             int fStable;
        } PrepareBlockType;


typedef
        struct {
             int sKey;
        } keyBlockType;

typedef
        struct {
             int KeyOffset;
             int KeyLength;
             int KeyType;

             int fAscending;

        } SortKeyType;

typedef
       struct {
            int cSortKeys;
            SortKeyType pSortKeys[ 1 ];
       } KeyDescType;
```

APPENDIX C: REMOVING TEMPORARY SORT/MERGE WORK FILES MANUALLY

If Sort/Merge terminates abnormally (that is, TerminateSort does not run), the application programmer must manually remove the temporary work files used by Sort/Merge. To remove these files:

1. Move to the directory that contains the temporary work files by entering the following command:

   ```
   cd /usr/temp
   ```

2. Display the names of the Sort/Merge temporary work files using the UNIX list command (ls -a). Not all files in this list are necessarily Sort/Merge files. Sort/Merge file names typically begin with a period and two letters followed by some digits (e.g., .AA01205 is a temporary work file name).

3. Remove all Sort/Merge temporary work files, using the UNIX remove command (rm).

APPENDIX D: DEFINING A NON-ASCII COLLATING SEQUENCE

The default collating sequence for comparing alphanumeric keys in an AT&T **UNIX** PC Sort/Merge operation is ASCII. However, a non-ASCII collating sequence, such as EBCDIC, can be defined for character-type keys. There are two ways to define a new collating sequence:

1. Within the application program
2. By customizing Sort/Merge.

The two methods are described below.

DEFINING A COLLATING SEQUENCE WITHIN THE APPLICATION PROGRAM

To define a non-ASCII collating sequence in the application program:

1. Assign the address of the new collating sequence table to the tblCollSeq variable. This variable must be declared as an external address variable in the application program.
2. The application program will subsequently reference the user's collating sequence table, instead of the ASCII table, only for comparing all character-type keys.

See the example on the next page.

Defining a Non-ASCII Collating Sequence

```
/*      This table defines the default calling sequence for types
        of CHARACTER

        This table should be modified if different calling sequence
        is desired.
*/

extern unsigned char *tblCollSeq;
unsigned char altCollSeq[]=  {0,1,2,3,4,5,6,7,8,9
                              10,11,12,13,14,15,16,17,18,19,
                              20,21,22,23,24,25,26,27,28,29,
                              30,31,32,33,34,35,36,37,38,39,
                              40,41,42,43,44,45,46,47,48,49,
                              50,51,52,53,54,55,56,57,58,59,
                              60,61,62,63,64,97,98,99,100,101,
                              102,103,104,105,106,107,108,109,110,111,
                              112,113,114,115,116,117,118,119,120,121,
                              122,91,92,93,94,95,96,97,98,99,
                              100,101,102,103,104,105,106,107,108,109,
                              110,111,112,113,114,115,116,117,118,119,
                              120,121,122,123,124,125,126,127,128,129,
                              130,131,132,133,134,135,136,137,138,139,
                              140,141,142,143,144,145,146,147,148,149,
                              150,151,152,153,154,155,156,157,158,159,
                              160,161,162,163,164,165,166,167,168,169,
                              170,171,172,173,174,175,176,177,178,179,
                              180,181,182,183,184,185,186,187,188,189,
                              190,191,192,193,194,195,196,197,198,199,
                              200,201,202,203,204,205,206,207,208,209,
                              210,211,212,213,214,215,216,217,218,219,
                              220,221,222,223,224,225,226,227,228,229,
                              230,231,232,233,234,235,236,237,238,239,
                              240,241,242,243,244,245,246,247,248,249,
                              250,251,252,253,254,255};

In body of program:
tblCollSeq = altCollSeq;
```

CUSTOMIZING SORT/MERGE TO ACCOMMODATE A NEW COLLATING SEQUENCE

To customize Sort/Merge to accommodate a new collating sequence:

1. Create a new collseq.c file. Change the numbers in the tblCollSeq table defined in the collseq.c file to conform to the needed collating sequence. The following listing shows the contents of the collseq.c file in which the tblCollSeq table is defined for the ASCII collating sequence.

2. Type the following commands from the shell:

```
su (enter password, if any)
cc -c collseq.c
ar -r /usr/lib/libsort.a collseq.o
exit
```

   All character sorting will now use the new sequence.

```
/*    This table defines the default calling sequence for types of
      CHARACTER

      This table should be modified if different calling sequence is
      desired.
*/

unsigned char tblCollSeq[]=  {0,1,2,3,4,5,6,7,8,9
                              10,11,12,13,14,15,16,17,18,19,
                              20,21,22,23,24,25,26,27,28,29,
                              30,31,32,33,34,35,36,37,38,39,
                              40,41,42,43,44,45,46,47,48,49,
                              50,51,52,53,54,55,56,57,58,59,
                              60,61,62,63,64,97,98,99,100,101,
                              102,103,104,105,106,107,108,109,110,111,
                              112,113,114,115,116,117,118,119,120,121,
                              122,91,92,93,94,95,96,97,98,99,
                              100,101,102,103,104,105,106,107,108,109,
                              110,111,112,113,114,115,116,117,118,119,
                              120,121,122,123,124,125,126,127,128,129,
                              130,131,132,133,134,135,136,137,138,139,
                              140,141,142,143,144,145,146,147,148,149,
                              150,151,152,153,154,155,156,157,158,159,
                              160,161,162,163,164,165,166,167,168,169,
                              170,171,172,173,174,175,176,177,178,179,
                              180,181,182,183,184,185,186,187,188,189,
                              190,191,192,193,194,195,196,197,198,199,
                              200,201,202,203,204,205,206,207,208,209,
                              210,211,212,213,214,215,216,217,218,219,
                              220,221,222,223,224,225,226,227,228,229,
```

```
                                230,231,232,233,234,235,236,237,238,239,
                                240,241,242,243,244,245,246,247,248,249,
                                250,251,252,253,254,255};
```

APPENDIX E: PROGRAM EXAMPLES

This appendix provides examples of AT&T **UNIX** PC BASIC, C, SVS™ FORTRAN, and SVS™ Pascal programs that call **UNIX** PC Sort/Merge procedures to sort data files.

The Sort/Merge facility is written in C and, therefore, conforms to C parameter passing standards, which are different from either FORTRAN or Pascal's passing standards. Consequently, to call Sort/Merge from either a FORTRAN or Pascal program, it is necessary to call a assembly program interface, called a "wrapper" routine, to act as an intermediary. The examples of Sort/Merge usage from FORTRAN and Pascal provided in this appendix illustrate the use of the appropriate assembly language wrappers.

BASIC EXAMPLE

This example demonstrates the use of **UNIX** PC Sort/Merge in a BASIC program to sort a data file.

The data declaration area, found in lines 20-50, defines the size, type, and shape of each of the data structures to be used in the program. The OPTION BASE statement is necessary, because the program assumes that the first elements of the arrays KYDSC and PRPRBLK are in subscript 1. Unless the OPTION BASE statement appears in this form, BASIC will assume that the lower bound of an array is zero. The DEFINT statement declares that all variables in the program are to be of type integer. Integers are 32 bits (four bytes) long, which makes it easy to conform to the rules of the Sort/Merge data structures.

The initialization section of the program is found in statements 60-200. The program opens the input and output files and sets the initial values in the PrepareKeySort data structures. Note that in statement 190 the address of the two parameters is passed using the BASIC PTR function, which passes the address of the array, rather than the values of the array itself.

The records are read and passed to the Sort/Merge package in lines 240-300. Note that the BASIC LEN function is used to get the length of the line to be sorted.

The records are sorted in line 340 by DoSort.

The sorted records are returned to the program in lines 385-440. Since the variable LINE$ is a string variable, space for its value is normally allocated by BASIC when a BASIC statement assigns the value, such as during a LET statement or an INPUT statement. But since the value is assigned by ReturnRecord outside of the BASIC subsystem, adequate space for LINE$ must be allocated ahead of time. This is done in statement 390. If this step is omitted, serious internal errors in BASIC will result.

Final cleanup is performed in statements 480-500.

The Sort/Merge utilities can be called from BASIC, only after it has been customized to contain the definitions of those utilities. To customize BASIC, you must enable all six definitions under Sort/Merge in BASIC's Basgen.config file by removing the comment symbols before and after the definitions. To perform this operation, you must be logged on as root and be in the directory /usr/lib/basic, where Basgen.config and the other files are located. Before you edit Basgen.config, the definitions look like

```
   /* S o r t / M e r g e  p a c k a g e */

 /* int = PrepareKeySort( &int, &int )    */
 /* int = ReleaseRecord( int, &int)       */
 /* int = DoSort                          */
 /* int = ReturnRecord( &short, &int )    */
 /* call ConcludeSort                     */
 /* call TerminateSort                    */

  /* S o r t / M e r g e  p a c k a g e */

   int = PrepareKeySort( &int, &int )
   int = ReleaseRecord( int, &int)
   int = DoSort
   int = ReturnRecord( &short, &int )
   call ConcludeSort
   call TerminateSort
```

Customizing the Interpreter

To call Sort/Merge from the BASIC Interpreter, after editing Basgen.config, enter the following commands to rebuild the interpreter:

```
su (enter password, if any)
/usr/lib/basic/build Basgen.config
cc -o /usr/bin/basic basic.o Call.c -lsort -lm
```

Customizing the Compiler

To call Sort/Merge from a compiled BASIC program, after editing Basgen.config, enter the following commands to rebuild the compiler:

```
su (enter password, if any)
/usr/lib/basic/cbuild Basgen.config
cc -o /usr/bin/bcrun bcrun.o Call.c -lsort -lm
```

For complete details on how to customize BASIC, refer to the section on "Customizing BASIC," in your BASIC language manual.

```
20 OPTION BASE 0
30 DEFINT A-Z
40 DIM  KYDSC[ 5 ]
50 DIM PRPRBLK[ 2 ]
60 INPUT "Enter name of file to be sorted:" ; FILENAME$

70 OPEN  "i", #1, FILENAME$
80 INPUT "Enter name of file to be written to:" ; OUTPUTFILE$
90 OPEN "o", #2, OUTPUTFILE$
100 PRPRBLK[ 0 ] = 80          ' Maximum record size
105 PRPRBLK[ 1 ] = 1           ' Stable sort wanted
110 KYDSC[ 0 ] = 1             ' number of sort keys is 1
115 KYDSC[ 1 ] = 3             ' key offset is 3
120 KYDSC[ 2 ] = 8             ' length of key is 8
130 KYDSC[ 3 ] = 1             ' Key type is 1 (bytes)
140 KYDSC[ 4 ] = 1             ' want an ascending sort
170 RECORDCOUNT = 0
180 PRINT  "Sorting..."
190 ERROR% =  PrepareKeySort[ PTR( PRPRBLK ), PTR( KYDSC ) ]
200 IF ERROR% < 0  THEN PRINT "Couldn't process this key"  : CALL
TerminateSort  : STOP
210 '
220 ' loop to read and sort the records
230 '
240 RECLEN = 0
245 ON ERROR GOTO 340
250 LINE INPUT #1, LINE$
260 IF LEN( LINE$ ) = 0%  THEN GOTO 340
270 RECORDCOUNT =  RECORDCOUNT + 1%
280 ERROR% =  ReleaseRecord( LEN( LINE$ ), PTR( LINE$ ) )
290 IF  ERROR% < 0%  THEN PRINT  "ERROR=  ";ERROR%: CALL
TerminateSort  : STOP
300 GOTO 250
310 '
320 ' When it gets here, it's made the first pass thru the file
330 '
340 ERROR% =  DoSort
350 IF ERROR% < 0  THEN PRINT  "Couldn't do the sort"  : CALL
TerminateSort  : STOP
360 '
370 ' Now get back the sorted file
```

```
380 '
385 FOR I = 1 TO RECORDCOUNT
390     LINE$ =  SPACE$( 80 )
400     ERROR% =  ReturnRecord[ PTR( RECLEN ), PTR( LINE$ ) ]
410     IF ERROR% > 0 GOTO 480
420     IF ERROR% < 0 THEN PRINT  "Error in ReturnRecord"  : STOP

430     PRINT #2, LINE$
440 NEXT I
450 '
460 ' clean up and quit
470 '
480 CALL ConcludeSort
490 PRINT "All done...."
500 END
```

C EXAMPLE

This example demonstrates the use of **UNIX** PC Sort/Merge in a C program to sort a data file.

The declaration section starts with the first line of the program and ends just before the comment, /* Initialize */. The "struct" statements for each of the Sort/Merge data structures have been taken directly from the Sort/Merge sources themselves, and the use of these declarations is recommended.

The initialization section extends from the /*Initialize */ comment to just before the Loop comment. Note that TerminateSort is called if an error occurs.

The loop section reads and sorts records.

The last section gets the sorted records back from the Sort/Merge package and outputs them to the screen. Final cleanup is performed by ConcludeSort.

When compiling a C program, be sure to include the sort and math libraries, as in:

```
cc filename.c -lsort -lm
```

```
/* Sort demonstration.  Sorts a single file and outputs the
sorted records */
{
#include <stdio.h>
#define TRUE 1
#define FALSE 0
#define NIL 0
main()

    typedef struct {
```

```
        int KeyOffset;
        int KeyLength;
        int KeyType;
        int fAscending;
    } SortKeyType;
    struct {
        int cSortKeys;
        SortKeyType SortKeys[ 1 ];
    } KeyDescriptor;
    struct {
        int MaxRecordLength;
        int fStable;
    } PrepareBlock;

    int Error;
    int fEOF;
    int i;

    int ch;

    char InputFileName[ 80 ];

    FILE *FP;
    char Record[ 80 ];
    short RecordLength;
    char buffer[ BUFSIZ ];

    int NumberOfRecords;


/* Initialize */


printf( "Enter name of file to be sorted: " );
scanf( "%s", InputFileName );
if ( ( FP = fopen( InputFileName, "r" ) ) == NIL ) {
    printf( "Couldn't open that file...\n" );
    exit( 1 );
};
setbuf( FP, buffer );

PrepareBlock.MaxRecordLength = 80;
PrepareBlock.fStable = TRUE;

KeyDescriptor.cSortKeys = 1;
KeyDescriptor.SortKeys[0].KeyOffset = 3;
KeyDescriptor.SortKeys[0].KeyLength = 8;
KeyDescriptor.SortKeys[0].KeyType = 1;
KeyDescriptor.SortKeys[0].fAscending = TRUE;

if ( ( Error = PrepareKeySort( &PrepareBlock, &KeyDescriptor ) ) < 0 )
    printf( "Couldn't process this key\n" );
    TerminateSort();
    exit( 1 );
```

```
};

NumberOfRecords = 0;
fEOF = FALSE;
/* Loop:  read records and give them to ReleaseRecord */

while ( ! fEOF ) {

    RecordLength = 0;
    while ( ( ch = getc( FP ) ) != '
        Record[ RecordLength++ ] = ch;
    fEOF = ( ch == EOF );

    if ( ! fEOF && RecordLength != 0 ) {
        NumberOfRecords++;
        if ( ( Error = ReleaseRecord( RecordLength, Record ) ) < 0 )
            printf( "Error in ReleaseRecord...\n" );
            TerminateSort();

            exit( 1 );
        };
    };
};

/* When it gets to here, all of the records have been read and passed

   ReleaseRecord.  Sort them. */

if ( ( Error = DoSort() ) < 0 ) {
    printf( "Error in the sort...\n" );

    TerminateSort();
    exit( 1 );
};

/* Now loop to get back all of the records in sorted order.  Print the

for ( i = 1; i <= NumberOfRecords; i++ ) {

    if ( ( Error = ReturnRecord( &RecordLength, Record ) ) < 0 ) {
        printf( "Error in ReturnRecord...\n" );
        TerminateSort();
        exit( 1 );
    };
    Record[ RecordLength ] = '\0';
}; printf( "%s\n", Record);

/* All done. Clean up and quit */

ConcludeSort();
exit( 0 );
}
```

Program Examples

FORTRAN EXAMPLE

The following FORTRAN example accesses Sort/Merge through the medium of an assembly language wrapper described below.

Note in the variable declaration section the arrays preblk(2) and keydes(5). The preblk array passes the parameters for "PrepareSortBlock" described in Table 2-1. The keydes array passes the KeyDescriptor values described in Tables 2-2 and 2-3.

The "start of program" section prompts the user for the name of the unsorted input file (which must be an ASCII file that exists at execution time) and the name of the output file (where sorted records will be written).

The "set prepare block" section specifies a maximum record size of 100 and a stable sort. (These values could be changed by the user if desired.)

In the "set key descriptor" section, elements of the keydes array are set to represent 1 key, an offset of 0, a length of 20, character type (see Table 2-4), and ascending order.

To increase the number of keys, the keydes array must be expanded by 4 elements for each additional key. For example, to allow for two keys, keydes must be expanded to 9 elements, keydes(1) is set to 2, and keydes(6) through (9) contain the second set of key descriptors.

The program is then ready to prepare the key sort. Note that the calling sequence is in the form of a function call to the assembly wrapper routine "Prep". The wrapper will obtain the parameter pointers (preblk and keydes) and pass them to Sort/Merge with a call to PrepareKeySort itself.

Then the input file is opened and the sort operation is carried out by successive calls to the wrapper routines that invoke the Sort/Merge procedures:

Termin	TerminateSort (if error encountered)
relr	ReleaseRecord
Dosort	DoSort
Retr	ReturnRecord

Finally, the records from "ReturnRecord" are written to the output file and "Consot" directs the wrapper to invoke ConcludeSort.

```
      program sorter
C variable declaration
      character*80      record, fname, outfil
      integer*4         keydes(5), preblk(2), reclen
      integer*2         error, feof, i, ch, norec
```

```
C start of program
      write (*,10)
10    format ('Enter name of file to be sorted:'$)
      read (*,100) fname
100   format (A)
      write (*,11)
11    format ('Enter name of file to be written to:'$)
      read (*,100) outfil

C set prepare block
C maximum record size
      preblk(1)=100
C 1=stable sort, 0= not stable
      preblk(2)=1

C set key descriptor
C no. of keys in record
      keydes(1)=1
C keyoffset
      keydes(2)=0
C keylength
      keydes(3)=20
C keytype
      keydes(4)=2
C fAscending = 1 , descending = 0
      keydes(5)=1

C prepare key sort

      norec=0

      error= Prep(preblk,keydes)
      if (error .lt. 0) then
                write (*,20)
20              format ('Failed in PrepareKeysort')
                call Termin
                goto 2000
      endif

C open file
      open (3,File=fname)

C set the record length to 80
      reclen=80
C read from file, keep looping until ends
110   read (3,100,End=200) record

C pass the record to sort
      error=Relr(reclen,record)
      if (error .lt. 0)then
                write (*,150)
150             format ('Failed in ReleaseRecord')
                call Termin
                goto 2000
```

```
        endif

C clear record
        norec = norec + 1
        Do 180 i = 1, 80
180     record(i:i)=' '

C go back and read another record
        goto 110

200     continue

C do sort
        error=Dosort()

        if (error .lt. 0) then
                write (*,220)
220             format ('Failed in DoSort')
                call Termin
                goto 2000
        endif

C sort completed, write to output file
        open (4,File=outfil)
        Do 300 i = 1, norec
                error=Retr(reclen,record)
                if (error .lt. 0) then
                        write (*,240)
240                     format ('Failed in ReturnRecord')
                        call Termin
                        goto 2000
                endif

                write (4,250) record
250             format (80A)

                reclen=0

300     continue
2000    call Consot
        End
```

Assembly Wrapper

A wrapper is needed because FORTRAN passes all values by reference while the Sort/Merge C functions pass some parameters by value. Also, C parameters are pushed onto the stack in the reverse order from those of FORTRAN. The following assembly listing provides compatibility between the FORTRAN program and Sort/Merge.

Note in the global declarations that the FORTRAN routine names are in uppercase. FORTRAN provides all external names in uppercase regardless of the case used in the source program. Thus, prep, Prep, and PREP in FORTRAN all refer to the routine PREP.

The PREP routine works as follows:

1. Initially, FORTRAN places a pointer to preblk in stack + 0 and a pointer to keydes in stack + 4.

2. The "link" saves the frame pointer and sets the stack pointer to stack + 16.

3. The first "mov.l" moves the keydes pointer to stack + 16.

4. The next "mov.l" moves the preblk pointer to stack + 20.

5. The arguments are now in place to be accessed by PrepareKeySort, which is invoked by a jsr.

6. To prepare for the return, unlk transfers the frame pointer to the stack pointer and retrieves the original frame pointer.

7. The next two instructions move the return address from stack + 8 to stack + 0 and readjust the stack pointer to point to it.

```
        text
        global  PREP
        global  PrepareKeySort
        global  TERMIN
        global  TerminateSort
        global  RELR
        global  ReleaseRecord
        global  DOSORT
        global  DoSort
        global  RETR
        global  ReturnRecord
        global  CONSOT
        global  ConcludeSort
PREP:
        link    %a6,&-4
        mov.l   8(%a6),(%sp)
        mov.l   12(%a6),-(%sp)
        jsr     PrepareKeySort
        unlk    %a6
        mov.l   (%sp),8(%sp)
        add.l   &8,%sp
        rts
TERMIN:
        jsr     TerminateSort
```

```
        rts
RELR:
        link        %a6,&-4
        mov.l       10(%a6),(%sp)
        mov.l       14(%a6),%a0
        mov.l       (%a0),-(%sp)
        jsr         ReleaseRecord
        unlk        %a6
        mov.l       (%sp),10(%sp)
        add.l       &10,%sp
        rts
DOSORT:
        jsr         DoSort
        rts
RETR:
        link        %a6,&-4
        mov.l       10(%a6),(%sp)

        mov.l       14(%a6),-(%sp)
        jsr         ReturnRecord
        unlk        %a6
        mov.l       (%sp),10(%sp)
        add.l       &10,%sp
        rts
CONSOT:
        jsr         ConcludeSort
        rts
```

Other routines operate in a similar manner. RELR obtains the record pointer and reclen value for ReleaseRecord. RETR obtains both reclen and record as pointers, because the reclen pointer became known in RELR. TERMIN, DOSORT, and CONSOT have no arguments and are executed by jsr and rts.

Compiling and Linking

Assume that the FORTRAN program is named "sorter.for" and the wrapper is named "wrapfor.s". They can be compiled and linked as follows:

```
cc -c wrapfor.s
fortran sorter.for wrapfor.o /usr/lib/libsort.a /lib/libm.a
```

PASCAL EXAMPLE

The following Pascal example accesses Sort/Merge through the medium of an assembly language wrapper described below.

In the type declaration section, SortKeyType is established as a record containing the key component descriptors of Table 2-3. KeyDescriptor is set up as a record that includes an array of SortKeyType records. To increase the number of key fields, extend the array size from [1..1] to [1..n]. PrepareBlock is defined as a record containing the values in Table 2-1.

The Var section sets up various pointers and defines the actual Sort/Merge procedures as external procedures and functions. These are actually calls to the wrapper (described below) which obtains the parameters and invokes the appropriate Sort/Merge procedures.

Procedure Cleanup is defined for use in several places to put blanks in the character buffer, recbuf. Otherwise, surplus characters would be appended to short strings in the sorted output.

The main program then begins. It prompts the user for the name of the unsorted input file (which must be an ASCII file that exists at execution time). The file is opened, the character buffer is cleared out, and the sort parameters are established. In this example, parameters are set up for a record length of 80, a stable sort, one set of keys, an offset of 0, a keylength of 8, and an ASCII sort in ascending order. The user could change these values if desired. Remember that if the number of keys is changed, the "sortkeys" array subscripts must be altered accordingly. Also, if the keylength is changed, every record in the input file must have that number of characters or more. Otherwise, a runtime error occurs.

The program is then ready to prepare the key sort. Note that the calling sequence is in the form of a function call to the assembly wrapper routine "Prep". The wrapper will obtain the parameter pointers (p.maxrecordlength and k.sortkeys from the PrepareBlock record) and pass them to Sort/Merge with a call to PrepareKeySort itself.

Then the sort operation is carried out by successive calls to the wrapper routines that invoke the Sort/Merge procedures:

```
Termin      TerminateSort (if error encountered)
Relr        ReleaseRecord
Dosort      DoSort
```

Retr ReturnRecord

Finally, the records from "ReturnRecord" are written to the standard output and "Consot" directs the wrapper to invoke ConcludeSort.

```
        program Sorter;

        Type

        SortKeyType = record
                        keyoffset:  longint;
                        keylength:  longint;
                        keytype:  longint;
                        fascending:  longint;
                        end;

        KeyDescriptor = record
                        csortKeys:  longint;
                        sortKeys:  array [1...1] of SortKeyType;
                        end;
        PrepareBlock = record
                        maxrecordlength:  longint;
                        fstable:  longint;
                        end;

        pctype = packed array [1..80] of char;

        Var
        error, feof, i, recno : integer;
        reclen : longint;

        p : PrepareBlock;
        k : KeyDescriptor;

        f : packed file of char;
        recbuf: array [1..1] of pctype;
        ch : char;
        filename : string[80];

(* declare sort/merge routines to be external  *)

        Procedure Termin; external;
        Procedure Consot; external;
        Function Dosort:integer; external;
        Function Prep(var p,k : longint): integer; external;
        Function Relr(len: longint;
                var rec: pctype) :integer; external;
        Function Retr(var len: longint;
                var rec: pctype) :integer; external;

Procedure Cleanup;
Var
        i  :  integer;
Begin
```

```
        for i := 1 to 80 do
                recbuf[1][i]   := ' ';

End;

Begin

        writeln ('Enter name of file to be sorted:');
        readln (filename);
(* Turn off I/O checking  *)
(* $I-  *)
(* Open file  *)

        Reset (f, filename);
        if IORESULT <> 0 then begin
           writeln ('File does not exist!');
           HALT;
        end;
(* Clean up the recbuf  *)
        Cleanup;
        recno := 0;
        reclen := 1;

(* Setup sort routine data block *)
        p. maxrecordlength := 80;
        p.fStable := 1;

        k.csortKeys := 1;
        k.SortKeys[1].keyoffset := 0;
        k.SortKeys[1].keylength := 8;
        k.SortKeys[1].keytype := 2;
        k.SortKeys[1].fascending := 1;
(*  prepare sort  *)

        error := Prep(p.maxrecordlength, k.csortKeys);
        if (error < 0) then
        begin
           writeln ('Error in PrepareKeySort!');
           Termin;
           HALT;
        end;

(* If file exists and not empty, read until end of line  *)
(* read file until end-of-line detected  *)
           while ( not EOF(f)) do
           begin
              ch := f^;
              (* check for new line  *)
              if (EOLN(f)) then
              begin
                error  := Relr(reclen,recbuf[1]);
                if (error < 0) then
                begin
                   writeln ('Error in ReleaseRecord.');
```

```
                    Termin;
                    HALT;
                  end;
                  recno := recno + 1;
                  reclen :=1;
                  Cleanup;
                end
                else begin
                  recbuf[1][reclen] := ch;
                  reclen := reclen +1;
                end;
                get(f);
             end;

(*  all records have been sent to sort and now do the sorting)

       error := Dosort;
       if (error < 0) then
       begin
          Writeln ('Error in DoSort.');
          Termin;
          HALT;
       end;

(*  sort is completed and get records back from sorter  *)
       for i  := 1 to recno do

       begin
          Cleanup;
          error := Retr(reclen,recbuf[1]);
          if (error < 0) then
          begin
             Writeln ('Error in ReturnRecord.',error);
             Termin;
             HALT;
          end;

          writeln (recbuf[1]);
       end;

(*  Everything is complete and conclude sort  *)

       Consot;

End.
```

Assembly Wrapper

A wrapper is needed because C parameters are pushed onto the stack in the reverse order from those of Pascal, and some data types use a different number of bytes. The following assembly listing provides compatibility between the Pascal program and Sort/Merge.

Note in the global declarations that the Pascal routine names are in uppercase. Pascal provides all external names in uppercase regardless of the case used in the source program. Thus, prep, Prep, and PREP in Pascal all refer to the routine PREP.

The PREP routine works as follows:

1. Initially, Pascal places a pointer to the PrepareBlock record in stack + 0 and a pointer to the KeyDescriptor record in stack + 4.

2. The "link" saves the frame pointer and sets the stack pointer to stack + 16.

3. The first "mov.l" moves the KeyDescriptor pointer to stack + 16.

4. The next "mov.l" moves the PrepareBlock pointer to stack + 20.

5. The arguments are now in place to be accessed by PrepareKeySort, which is invoked by a jsr.

6. To prepare for the return, unlk transfers the frame pointer to the stack pointer and retrieves the original frame pointer.

7. The next two instructions move the return address from stack + 8 to stack + 0 and readjust the stack pointer to point to it.

```
        text
        global  PREP
        global  PrepareKeySort
        global  TERMIN
        global  TerminateSort
        global  RELR
        global  ReleaseRecord
        global  DOSORT
        global  DoSort
        global  RETR
        global  ReturnRecord
        global  CONSOT
        global  ConcludeSort
PREP:
        link    %a6,&-4
        mov.l   8(%a6),(%sp)
        mov.l   12(%a6),-(%sp)
        jsr     PrepareKeySort
        unlk    %a6
        mov.l   (%sp),8(%sp)
        add.l   &8,%sp
        rts
TERMIN:
```

```
        jsr     TerminateSort
        rts
RELR:
        link    %a6,&-4
        mov.l   8(%a6),(%sp)
        mov.l   12(%a6),-(%sp)
        jsr     ReleaseRecord
        unlk    %a6
        mov.l   (%sp),8(%sp)
        add.l   &8,%sp
        rts
DOSORT:
        jsr             DoSort
        rts
RETR:
        link            %a6,&-4
        mov.l           8(%a6),(%sp)
        mov.l           12(%a6),-(%sp)
        jsr             ReturnRecord
        unlk            %a6
        mov.l           (%sp),8(%sp)
        add.l           &8,%sp
        rts
CONSOT:
        jsr             ConcludeSort
        rts
```

Other routines operate in a similar manner. RELR obtains the record pointer and reclen value for ReleaseRecord. RETR obtains both reclen and record as pointers, because the reclen pointer became known in RELR. TERMIN, DOSORT, and CONSOT have no arguments and are executed by jsr and rts.

This wrapper assumes that "recordlength" in the source program is defined as long integer.

Compiling and Linking

Assume that the Pascal program is named "sort.pas" and the wrapper is named "wrappas.s". They can be compiled and linked as follows:

```
cc -c wrappas.s
pascal sort.pas wrappas.o /usr/lib/libsort.a /lib/libm.a
```

INDEX

A

B

C

C (continued)

D

E

F, G, H

O

P, Q

R

T, U, V

W, X, Y

Z