WANG

Sort Statements Reference Manual

Sort Statements Reference Manual

2nd Edition — January, 1981

700-3559G

WANG LABORATORIES, INC.
ONE INDUSTRIAL AVENUE, LOWELL, MASSACHUSETTS 01851, TEL. (617) 459-5000, TWX 710 343-6769, TELEX 94-7421

Disclaimer of Warranties and Limitation of Liabilities

The staff of Wang Laboratories, Inc., has taken due care in preparing this manual; however, nothing contained herein modifies or alters in any way the standard terms and conditions of the Wang purchase, lease, or license agreement by which this software package was acquired, nor increases in any way Wang's liability to the customer. In no event shall Wang Laboratories, Inc., or its subsidiaries be liable for incidental or consequential damages in connection with or arising from the use of the software package, the accompanying manual, or any related materials.

NOTICE:

All Wang Program Products are licensed to customers in accordance with the terms and conditions of the Wang Laboratories, Inc. Standard Program Products License; no ownership of Wang Software is transferred and any use beyond the terms of the aforesaid License, without the written authorization of Wang Laboratories, Inc., is prohibited.

WANG LABORATORIES, INC.
ONE INDUSTRIAL AVENUE, LOWELL, MASSACHUSETTS 01851, TEL. (617) 459-5000, TWX 710 343-6769, TELEX 94-7421

HOW TO USE THIS MANUAL

The Sort statements provide programmable, high-speed data sorting, searching, and moving capability to the Wang system. This manual describes the syntax and use of the Sort Statements of the BASIC instruction set. The Sort statement sets for the 2200VP/MVP and 2200VS CPU's are described in separate manuals.

It is assumed that the reader is familiar with the operation of the available system and has a working knowledge of the Wang BASIC language, or has access to Wang reference literature (e.g., the Wang BASIC Language Reference Manual, the Programming in BASIC Manual, the Disk Reference Manual, etc). A general knowledge of data processing techniques is also helpful.

Chapter 1 contains an overview of array fundamentals. Chapter 2 discusses each Sort statement in detail. It is recommended that novice programmers rely upon the Wang-supported sorting utilities rather than programming with the Sort statements. Users who wish to sort disk data files can refer to the ISS (Integrated Support System) sorting routines. Under certain conditions, a user may not need to prepare sorted files, but merely produce lists of sorted data. Another Wang-supported utility, KFAM (acronym for Keyed File Access Method), provides this capability. (See the KFAM Reference Manuals).

TABLE OF CONTENTS

CHAPTER 1 GENERAL INFORMATION

1.1 INTRODUCTION

The six Sort statements can be separated into two categories: those which are used in specific phases of a data-sorting program to reduce both the sort time and program size (MAT CONVERT, MAT SORT, MAT MOVE, MAT MERGE), and those which are generally useful in many text-editing, statistical and random access data retrieval applications to move or search character or data strings (MAT COPY and MAT SEARCH).

The Sort statements presented in this manual are:

MAT CONVERT — Reformats data from the internal numeric storage format into a hexadecimal sort format and stores it in an alphanumeric array.

MAT COPY — Copies data byte-by-byte from all or part of one alphanumeric array to all or part of another. Element boundaries are ignored and the data can be copied in forward or reverse directions.

MAT MERGE — Creates the locator-array used to merge more than one sorted file into a single sorted array or file.

MAT MOVE — Moves data from an alphanumeric input array to specified locations in an alphanumeric output array. The order of transfer is specified by a locator-array which contains the subscripts of the elements to be moved. Elements can be moved element-by-element, or by fields within each element.

MAT SEARCH — Searches an alphanumeric array for character strings that satisfy a given relationship. Element boundaries are ignored. The search can be performed at specified intervals and a variable number of characters can be compared.

MAT SORT — Sorts the elements of an alphanumeric array into ascending or descending order. Puts the subscripts of the ordered data in an output locator-array.

Since it is impossible to discuss all the sorting and merging techniques invented, examples have been chosen to illustrate sorting and the rules, syntax, and use of the Sort statements. The subscript locator-arrays (which are automatically created by MAT MERGE, MAT SEARCH and MAT SORT) are a means to access data by pointing to its location instead of moving it. Compared to the usual sort-and-exchange or bubble sort techniques, the ease of use, rapidity, and efficiency in memory utilization of "tag-sorting" mode excels. These features enhance the Wang system's ability to process applications where rapid access to large amounts of data is desirable, as in statistical analysis where it may be necessary to pass several times over the same mass of data and cull out information according to several parameters without dumping all the data each time. Using a BASIC program, it is fast and simple to read in an array, execute MAT SORT to order all elements in the array, and then execute MAT MOVE to place all the elements in the correct order on an output device.

1.2 INSTALLATION

The Sort statements contained in this manual are standard instructions in the BASIC language instruction set available on all Wang 2200T series CPU's (including WCS, PCS, and Work Stations). The Sort statements can usually be installed as an option in those Wang Series 2200 CPU's not already containing them.

1.3 ARRAYS

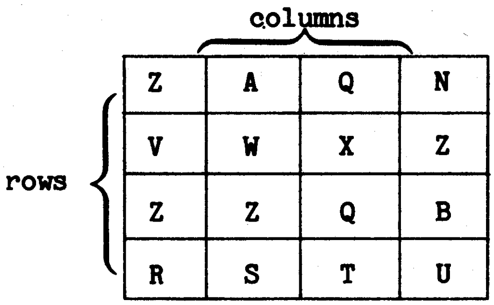
This section briefly describes the properties of arrays to assist the user of the Sort statements. Users already familiar with arrays and array notation can skip this section.

An array is a series of elements arranged in a specified order. Arrays can be either one- or two-dimensional. A one-dimensional array can be either a single row or a single column. A two-dimensional array is arranged in rows and columns. Because of this arrangement, the position of each element can be uniquely identified with subscripts. A one-dimensional array (also called a vector) has a single subscript. A two-dimensional array has two subscripts which always represent "row, column", never "column, row". A two-dimensional array resembles a table (see Figure 1-1 and Figure 1-2).

		columns	
	$r_{1,1}$	$r_{1,2}$	$r_{1,3}$
rows	$r_{2,1}$	$r_{2,2}$	$r_{2,3}$
	$r_{3,1}$	$r_{3,2}$	$r_{3,3}$

Figure 1-1. A Two-Dimensional Array

Both numeric and alphanumeric arrays can be manipulated with the Sort statements. Numeric and alphanumeric array elements are defined by the assigned array name, e.g., either A() or A$(), respectively. For example, in the following array the value in element (1,2) is A and the value in element (3,4) is B.



Z	A	Q	N
V	W	X	Z
Z	Z	Q	B
R	S	T	U

Figure 1-2. A Two-Dimensional Alphanumeric Array

The rules of the BASIC language require that every array be appropriately named and properly dimensioned with a DIM or COM statement prior to the variable's use in a program statement. This is required in order to reserve space for the array variables in memory. Space may be reserved for more than one array in a single dimension statement by separating the array names with commas. The space must be indicated explicitly; expressions are not allowed. The COM statement defines arrays which are to be used in common by several programs, a feature which is useful for program chaining. A COM statement must not change the dimensions of a previously defined common variable. If an array's dimensions are not specified in a DIM or COM statement the array will automatically be dimensioned as a 10 x 10 matrix. If the array is alphanumeric, each element in the array will have a length of 16 bytes. The following statement defines and dimensions the two-dimensional alphanumeric array illustrated in Figure 1-2.

```
10 DIM A$(4,4)
```

For alphanumeric arrays, the length of each element in the array can be specified by the programmer as any length between one byte and 64 bytes, inclusive. For example:

```
10 DIM A$ (4,4) 64
```

would specify a 16-element array with 64 bytes in each element (1024 bytes total).

The length of each element in the array, the total number of elements in the array, and the available memory size are the factors which limit the size of arrays which can be specified for use with the Sort statements. The system uses a single byte to represent each subscript of an array internally. Since the maximum binary number which can be represented in one byte (eight bits) is 255, each subscript of a two-dimensional array is restricted to a maximum value of 255. In other words, each row is limited to a maximum of 255 elements or each column is limited to a maximum of 255 rows. The maximum number of elements in any array is limited to 4096. The size of an array is also limited to the machine's total memory capacity.

> NOTE:
>
> Subscripts defined in DIM statements must be explicitly defined positive integers (1 to 255). Zero is not allowed as a defining subscript. Once defined, variables can be used as subscripts. In some Sort statements, a subset of an element can be defined. When memory space is initially allocated, (dimensioned) numeric array elements are set to zero, and alphanumeric array elements are set to spaces (HEX(20)).

1.4 INTERNAL STORAGE

All data is physically stored by the Wang System 2200 in a field of one or more contiguous bytes. The format in which data are stored internally depends upon whether the data are numeric values, alphanumeric values, or MAT CONVERTed numeric values. To identify each variable in memory, the system automatically inserts several bytes of control information at the beginning of the variable area. These additional bytes, however, are completely "transparent" to the programmer (i.e., they are used exclusively by the operating system and cannot be accessed by the application program). The number of control bytes required differs according to whether the array is alpha or numeric. Although control bytes represent a fixed overhead for each variable defined in a program, the amount of memory they require is insignificant.

Alphanumeric data are stored internally as a series of bytes. Each data element can range in length from one to sixty-four bytes, where each byte represents a single character.

A numeric value is stored internally in an eight-byte exponential form. The first half of the first byte contains a code for the relationship between the signs of the mantissa and the exponent. The next two half-bytes contain the exponent ($10^{-99} \leq E \leq 10^{+99}$) in reverse order (low-order digit, then high-order digit). The remaining 6 1/2 bytes contain the 13 digits of the mantissa. One digit is in each half-byte (see Figure 1.3). The standard numeric format is not good for sorting, however, because it does not describe a number's relative magnitude when reading consecutively across the byte field as is done during comparisons (sorting) when using the Sort statements. The MAT CONVERT statement restructures numeric data into a form suitable for sorting.

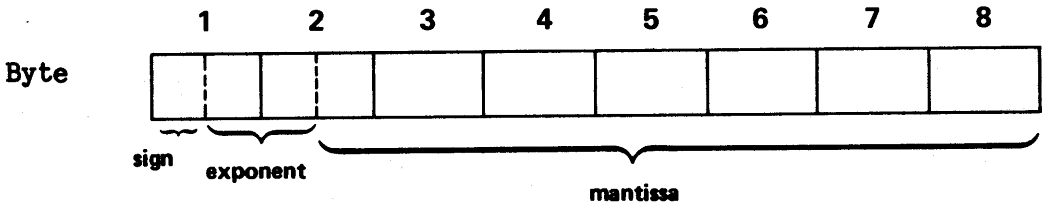


Figure 1-3. The Eight Byte Internal Numeric Storage Field

INTERNAL NUMERIC STORAGE RELATIONSHIPS

1st. half-byte	indicates	example
0	mantissa and exponent both positive	+1.0
8	mantissa positive, exponent negative	+0.1
9	mantissa and exponent both negative	-0.1
1	mantissa negative, exponent positive	-1.0

A MAT CONVERTed value is also stored in an exponential field (see Figure 1.3). However, the format of a MAT CONVERTed value is alphanumeric, not the standard internal numeric format. The MAT CONVERT format restructures numeric values so that the relationships between a number's sign, exponent, and mantissa, which describe its relative magnitude, occur consecutively in the first bytes of the field.

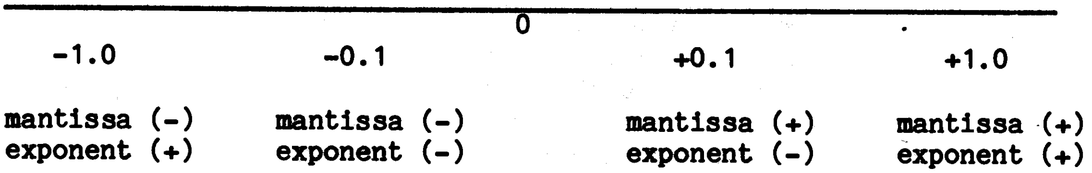
In the MAT CONVERTed format, the first half-byte represents the relationship between the signs of the mantissa and the exponent. The next two half-bytes represent the high-order and low-order digits of the exponent ($10^{-99} \leqslant E \leqslant 10^{+99}$) in their normal order. The exponent is given in decimal form if the signs of the mantissa and exponent are the same, or nines complement form if the signs of the mantissa and exponent differ. The remaining bytes of the value contain up to thirteen (13) digits of mantissa. These digits appear in decimal form if the sign of the mantissa is positive or in the nines complement decimal form if the sign of the mantissa is negative.

MAT CONVERTed STORAGE RELATIONSHIPS

1st half-byte	indicates	example
9	mantissa and exponent both positive	+1.0
8	mantissa positive, exponent negative	+0.1
1	mantissa and exponent both negative	-0.1
0	mantissa negative, exponent positive	-1.0

2nd and 3rd half-bytes	indicates	example
decimal	mantissa and exponent both positive	+1.0
complemented	mantissa positive, exponent negative	+0.1
decimal	mantissa and exponent both negative	-0.1
complemented	mantissa negative, exponent positive	-1.0

Using the five values of this number line example (A(1) = -1.0, A(2) = -0.1, A(3) = 0, A(4) = +0.1, A(5) = +1.0), Figure 1-4 compares the difference between the values when stored in the internal numeric storage format and MAT CONVERTed sort format.



INTERNAL NUMERIC FORMAT	MAT CONVERTed FORMAT
A(1) 1001000000000000	A$(1) 0998999999999999
A(2) 9101000000000000	A$(2) 1018999999999999
A(3) 0000000000000000	A$(3) 8000000000000000
A(4) 8101000000000000	A$(4) 8981000000000000
A(5) 0001000000000000	A$(5) 9001000000000000

Figure 1-4. Comparison of Internal Storage Formats

Notice how values are stored sequentially ascending in sort format, with each data element contained in a separate element of the alphanumeric array. Each element normally contains eight bytes. If the elements of the alpha array are specified to contain more than eight bytes, the stored value is padded with spaces (at the right). If the elements of the alpha-array are specified to contain less than eight bytes, the value is truncated; least significant digits are lost. The truncation feature can conserve memory when it is known that all values to be MAT CONVERTed will have fewer than 13 significant digits.

1.5 SORT/MERGE OPERATIONS

An internal sort is one in which all sorting occurs inside the memory of the computer. An external sort is performed when the memory is insufficient to contain the entire file to be sorted and various peripheral devices must be used for storage of permanent and intermediate data files. The most common sort or sort/merge techniques are performed externally.

Of the many sorting techniques which order data, a common technique which utilizes the MAT CONVERT, MAT COPY, MAT MOVE, MAT SORT and MAT MERGE statements is described below. This example contains a Convert Phase, Sort Phase, Merge Phase, and Move Phase, in that order.

The Sort key is the basis for ordering the data records. The Sort key can contain any number of variables or fields from the data record or even the entire record. MAT COPY is often used to concatenate data records and/or portions of data records to create the Sort key. Using a short Sort key improves running time because comparisons between the keys for sorting occur many times during a sort. In building the Sort key, note that both numeric and alphanumeric data must be compared in the same operation, character-by-character. Therefore, all data must be in sort format.

The Convert Phase

Numeric values are stored in a floating point exponential format which is optimum for numerical calculations. These numeric values must be converted for sorting purposes to an appropriate alphanumeric equivalent using MAT CONVERT. If the full thirteen digits of the mantissa are not required, MAT CONVERT can truncate the mantissa to conserve memory and shorten the Sort key.

The Sort Phase

In this phase, MAT SORT is used to sort the initial data into several ordered strings (small sorted groups) by setting up a subscript array (locator-array) whose elements point in sorted order to the locations of the records which are in memory. The data records to be sorted are ordered on the basis of a Sort key. The data records are read in groups, Sort keys are built for each group, and the records and/or keys are sorted into strings and stored on disk or tape in at least two Merge files.

Although the size of a string is not restricted by memory capacity, the number of records and/or keys initially put into each string is usually determined by the amount of memory available.

The Merge Phase

The Merge phase generally consists of a number of merge passes. In each merge pass, strings from two or more Merge files from the Sort phase are read, combined into longer strings, and stored in an output file. The process is repeated until a single string containing the entire file in sorted order is created. A given string need not be completely stored in memory at one time during the merge process; it can be loaded into memory in segments.

MAT MERGE creates the locator-array used to merge the input string segments. Each row of the merge-array represents one string segment from the input Merge file. The strings contain the records or keys on which comparisons are made. When MAT MERGE first executes, the first key in each row is compared with the first key in every other row, and the lowest key is selected. The subscript of the lowest key is placed in the locator-array and this key is eliminated from future comparisons. MAT MERGE is then executed again. The lowest key not yet merged is again sought. MAT MERGE execution terminates when a segment from the Merge file (a row) is exhausted or the locator-array is full.

The Move Phase

The Move phase consists of moving data from the merge-array to an output-array in the order specified by the locator-array. MAT MOVE transfers the merging records into an array where segments of the output file are being collected. Once a merge-array input segment has been exhausted, the row can be refilled using MAT COPY. This process is repeated until all strings from the Merge files have been exhausted and a single sorted string containing all records from the original input file has been created.

1.6 DESCENDING SORTS

Sometimes it is necessary to perform a sort in descending order. This can be accomplished by inverting the order of an ascending sort or by complementing (using XOR or BOOL6) the characters in the Sort key. For numeric data, complementing must be done after data have been converted to sort format using MAT CONVERT.

Consider the characters "A" (whose decimal and binary equivalents are 65 and 0100 0001 respectively) and "Z" (whose decimal and binary equivalents are 90 and 0101 1010 respectively). XOR performs the exclusive OR logical binary operation which outputs a 1-bit when the corresponding bits in the two arguments are different. For example:

```
     A = 0100  0001             Z = 0101  1010
    FF = 1111  1111            FF = 1111  1111
  -----------------         -----------------
   XOR = 1011  1110           XOR = 1010  0101
```

The complement of the code for "A" (whose decimal and binary equivalents are now 190 and 1011 1110 respectively) will sort after the complement of the code for "Z" (whose decimal and binary equivalents are now 165 and 1010 0101 respectively).

Example 1:

A$() is a Sort key to be complemented.

```
100 XOR (A$(),FF)
```

places all Sort keys in reverse sequence.

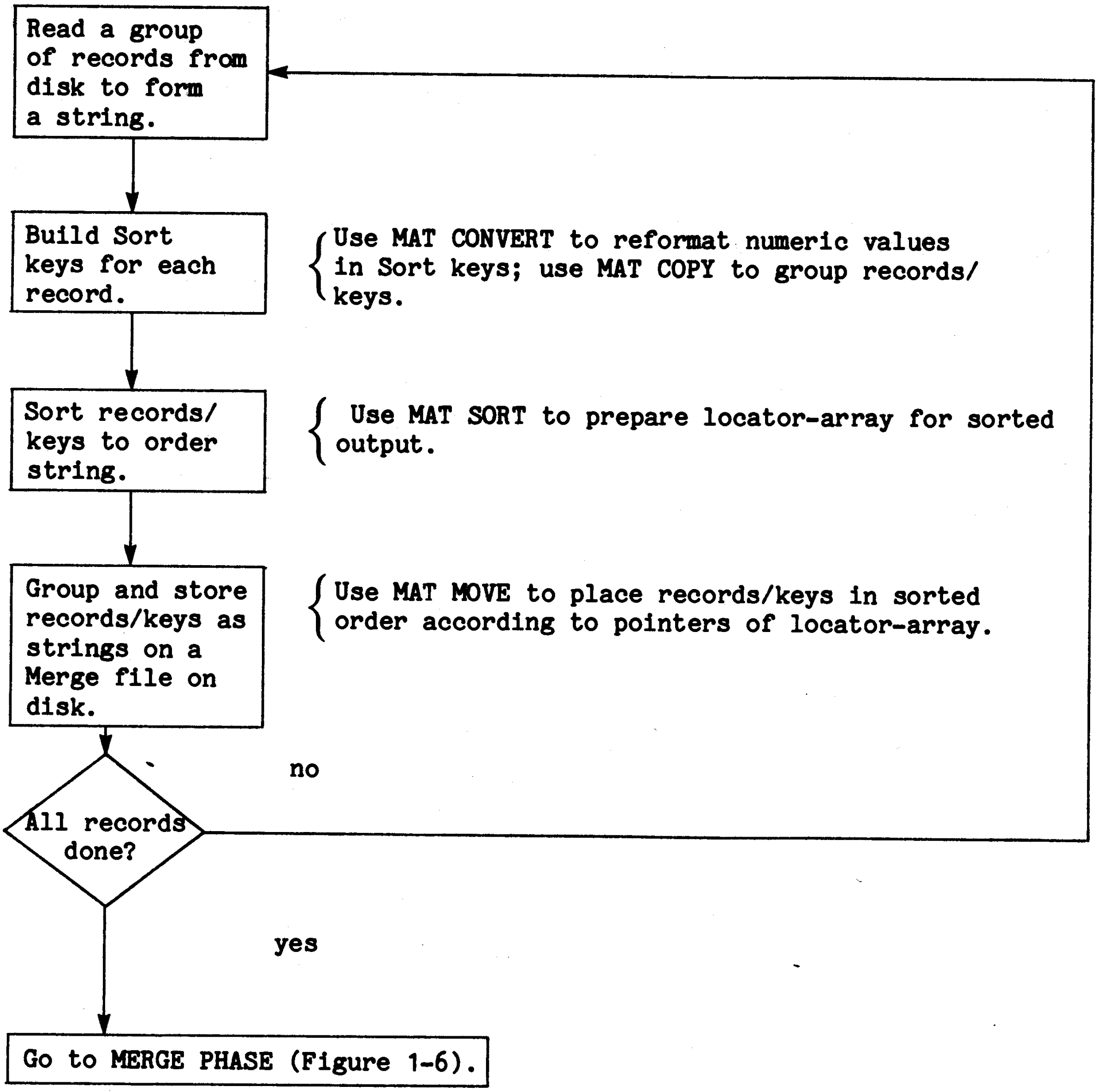
Example 2:

B$(1) is an array element representing a Sort key and record.The sixth through tenth characters of this element are part of the Sort key to be complemented for a descending sort.

```
200 XOR (STR(B$(1),6,5),FF)
```

complements the sixth through tenth characters of this Sort key.

Block diagrams for a typical program using Sort statements are shown in Figure 1-5 and Figure 1-6.


Read a group of records from disk to form a string.
Build Sort keys for each record.
Use MAT CONVERT to reformat numeric values in Sort keys; use MAT COPY to group records/keys.
Sort records/keys to order string.
Use MAT SORT to prepare locator-array for sorted output.
Group and store records/keys as strings on a Merge file on disk.
Use MAT MOVE to place records/keys in sorted order according to pointers of locator-array.
All records done?
no
yes
Go to MERGE PHASE (Figure 1-6).

NOTE:

Due to the many simultaneous operations that proceed when MAT MERGE executes, there are no simple examples to illustrate this statement.

Figure 1-5. Block Diagram of a Typical Sort Program (Sort Phase)

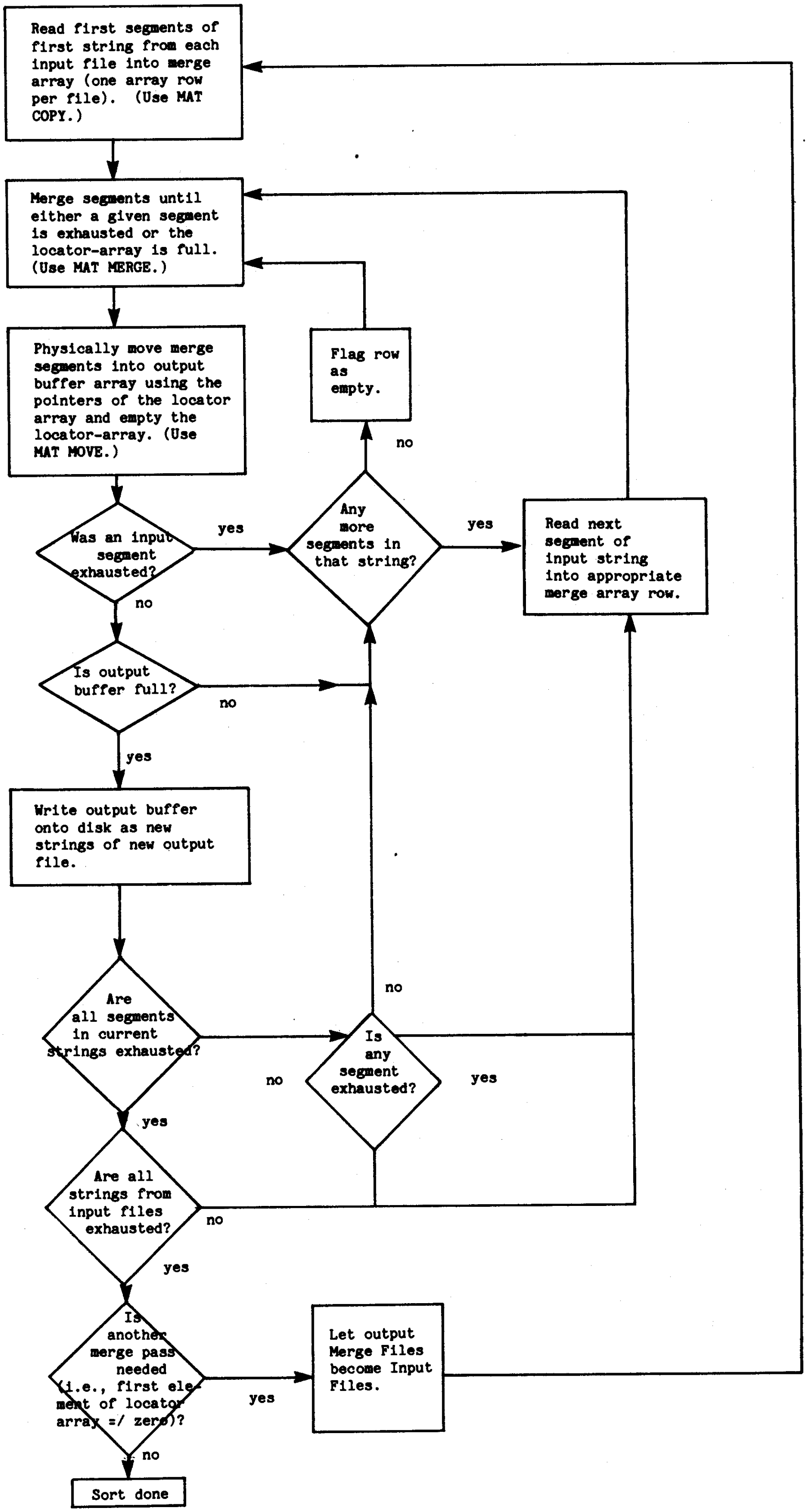

Read first segments of first string from each input file into merge array (one array row per file). (Use MAT COPY.)
Merge segments until either a given segment is exhausted or the locator-array is full. (Use MAT MERGE.)
Physically move merge segments into output buffer array using the pointers of the locator array and empty the locator-array. (Use MAT MOVE.)
Flag row as empty.
no
Was an input segment exhausted?
yes
Any more segments in that string?
yes
Read next segment of input string into appropriate merge array row.
no
Is output buffer full?
no
yes
Write output buffer onto disk as new strings of new output file.
no
Are all segments in current strings exhausted?
Is any segment exhausted?
no
yes
yes
Are all strings from input files exhausted?
no
yes
Is another merge pass needed (i.e., first element of locator array =/ zero)?
yes
Let output Merge Files become Input Files.
no
Sort done

Figure 1-6. Block Diagram of a Typical Sort Program (Merge Phase)

Example 3: An External Sort

In this example there are three sorted data files on disk (INPUT1, INPUT2 and INPUT3). Each data file has logical records which contain 50 elements; each element contains 8 bytes. The program merges the three files into a single file called output. The diagram illustrates the flow of data from the input files through the merge process, to the output buffer, and then to the output file.

The merging scheme proceeds as follows:

1. Data is brought into memory from the disk files utilizing an input buffer to transfer whole records at once.

2. A merge-array is created, M$(), using MAT COPY. In this example, MAT COPY is in a subroutine.

3. The first N elements of the primary work-array, W1$(), are initialized to 1.

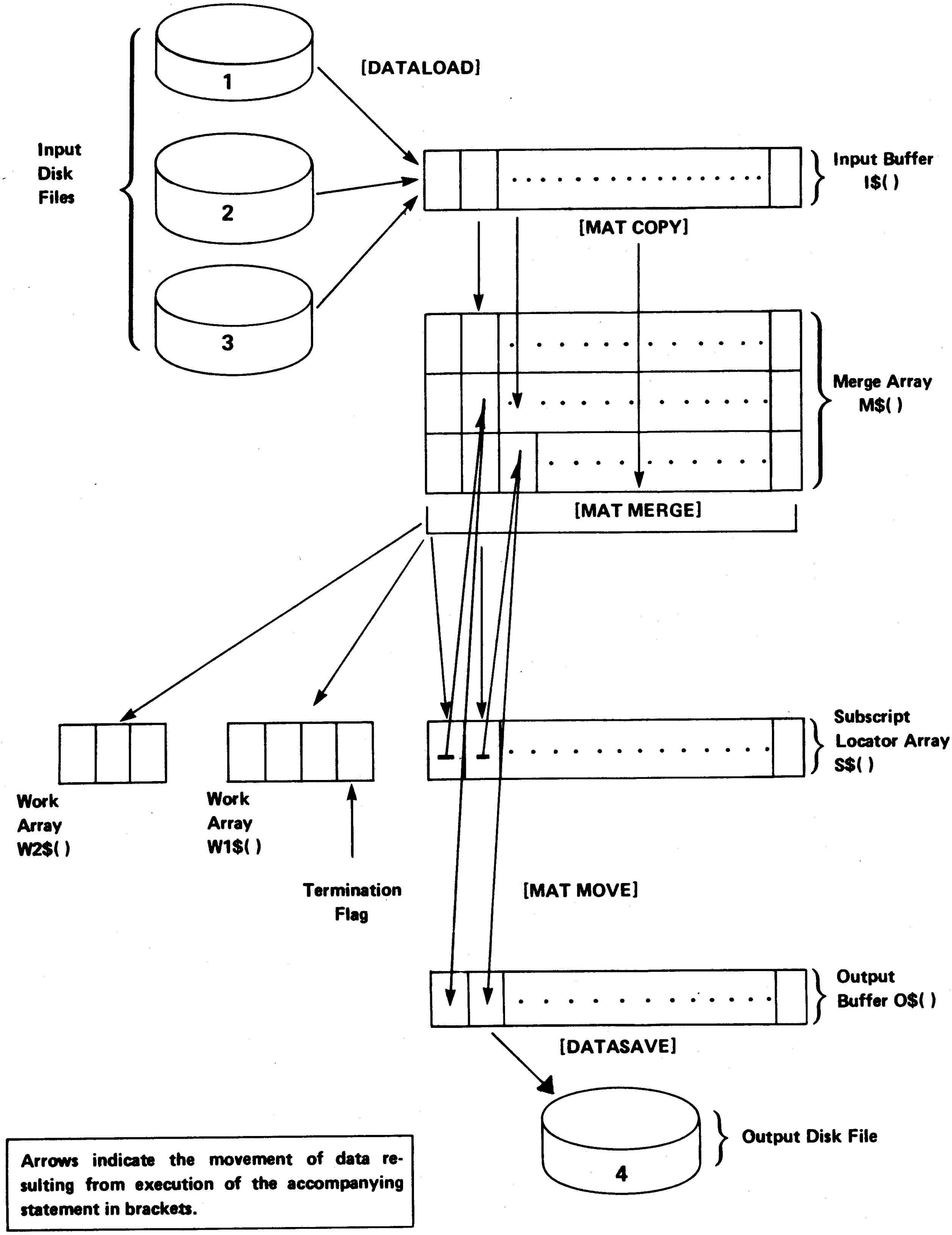
4. MAT MERGE is called to create the subscript locator-array.

5. MAT MOVE uses the locator-array to transfer data from the merge array to the output buffer. The movement of data ceases when the locator-array has been emptied or when an element filled with zeroes is reached. MAT MOVE automatically keeps track of the number of elements actually moved.

6. When the output buffer is filled, all data stored in it is written to the output file. The buffer is then ready to receive more data from another execution of MAT MOVE.

7. The last element in the primary work-array is checked (with the VAL function) to see if a row in the merge-array has been completely scanned. If all elements in a merge-array row have been used, the row is refilled with data from the input file and the appropriate element in the primary-work-array is reset to 1.

8. Steps 4 through 8 are repeated until the first element in the locator-array is filled with zeroes when examined by the MAT MOVE statement.



Arrows indicate the movement of data resulting from execution of the accompanying statement in brackets.

Figure 1-7. Schematic Diagram of MAT MERGE External Sort

Program Listing

```
10 REM ARRAY DEFINITIONS.................................................
20 DIM M$(3,50)8: REM MERGE ARRAY
30 DIM I$(50)8:   REM INPUT BUFFER
40 DIM O$(50)8:   REM OUTPUT BUFFER
50 DIM W1$(4)1:   REM WORK BUFFER 1
60 DIM W2$(3)2:   REM WORK BUFFER 2
70 DIM S$(50)2:   REM SUBSCRIPT BUFFER
80 REM OPEN THE DATA FILES ON DISK.......................................
90 SELECT #1 310, #2 310, #3 310, #4 310
100 DATA LOAD DC OPEN F #1, "INPUT1"
110 DATA LOAD DC OPEN F #2, "INPUT2"
120 DATA LOAD DC OPEN F #3, "INPUT3"
130 DATA LOAD DC OPEN F #4, "OUTPUT"
140 REM FILL THE MERGE ARRAY.............................................
150 FOR I = 1 TO 3
160 GOSUB '40(I)
170 NEXT I
180 REM INITIALIZE WORK BUFFER 1.........................................
190 INIT(01) W1$()
200 M=1
210 REM MERGE............................................................
220 MAT MERGE M$() TO W1$(), W2$(), S$()
230 IF S$(1)=HEX(0000) THEN 480: REM EXIT IF DONE
240 REM MOVE THE MERGED DATA TO THE OUTPUT BUFFER........................
250 S=1
260 N=50
270 MAT MOVE M$(), S$(S), N TO O$(M)
280 M=M+N: REM M = NO. OF ELEMENTS IN OUTPUT BUFFER
290 REM PUT DATA INTO OUTPUT FILE IF OUTPUT BUFFER FULL..................
300 IF M<=50 THEN 350
310 DATA SAVE DC #4, O$()
320 M=1
330 S=S+N
340 IF S<51 THEN 260: REM BRANCH IF MORE DATA TO MOVE
350 REM CHECK MERGE TERMINATION FLAG.....................................
360 T=VAL(W1$(4))
370 IF T=0 THEN 220: REM BRANCH IF NO ROWS OF M$() EMPTY
380 GOSUB '40(T): REM REFILL EMPTY ROW OF M$()
390 GOTO 220
400 REM..................................................................
410 DEFFN'40(R): REM READ THE NEXT BLOCK FROM SPECIFIED
420 REM              INPUT FILE AND PUT INTO MERGE ARRAY
430 DATA LOAD DC #R, I$()
440 IF END THEN 470
450 MAT COPY I$() TO M$()<(R-1)*400+1,400>
460 W1$(R)=HEX(01): REM RESET APPROPRIATE WORK BUFFER ELEMENT
470 RETURN
480 END
```

Example 4: An Internal Sort

In this example, two arrays, A$() and B$(), are internally merged. Their data are given in a DATA statement. Only a listing of this routine is provided, followed by a printout of the data which are displayed on the CRT. The routine uses MAT COPY to fill the input array, A$(); INIT to initialize the work vector, W1$(), and the locator-array, L$(); and MAT MERGE to fill the locator-array. Once L$() has been filled, the data are sent to an output array, O$(), with MAT MOVE. When the first element of the locator-array is still zero after an execution of MAT MERGE, the merge operation is considered complete and the final merged array is displayed. Not all arrays can be displayed directly with a PRINT statement. The HEXPRINT statement must be used for those arrays which cannot be displayed directly by a PRINT statement.

Program Listing

```
10 REM  THIS ROUTINE ILLUSTRATES USE OF MAT COPY, MAT MERGE
20 REM  AND MAT MOVE
30 REM        DEFINE THE ARRAYS
40 DIM A$ (4) 5, I$ (4) 5, B$ (4) 5
50 DIM M$ (2,4) 5,O$ (4) 5,W1$ (3) 1,W2$ (2) 2
60 DIM L$ (4) 2,J$ (9) 5,K$ (16) 5
70 S=1:R=1
80 REM         GET THE DATA TO BE MERGED
90 FOR I=1 TO 4:READ A$ (I) :NEXT I:FOR I=1 TO 4:PRINT A$ (I) ,:NEXT I
100 FOR I=1 TO 4:READ B$ (I) :NEXT I:FOR I=1 TO 4:PRINT B$ (I) ,:NEXT I
110 STOP "ALL DATA HAVE BEEN INPUT"
120 PRINT HEX (03)
130 PRINT "THIS IS THE ARRAY I$#1"
140 REM         FILL THE MERGE ARRAY M$ USING MAT COPY
150 FOR I=1 TO 4:I$ (I) =A$ (I) :PRINT I$ (I) ,:NEXT I:MAT COPY I$ () TO M$ ()
<1,20> :STOP
160 FOR I=1 TO 4:I$ (I) =B$ (I) :PRINT I$ (I) ,:NEXT I:PRINT" "
170 MAT COPY I$ () TO M$ () <21,20>
180 PRINT "THIS IS THE ARRAY I$#2":STOP
190 PRINT "HEX OF THE MERGE ARRAY M$,OUTPUT OF MAT COPY"
200 FOR I=1 TO 2:FOR J=1 TO 4:HEXPRINT M$ (I,J) :NEXT J:NEXT I:STOP
210 REM         INITIALIZE THE WORK VECTOR W1$ TO 1
220 INIT (01), W1$ ( )
230 PRINT HEX (03)
240 PRINT "THE MERGE ARRAY HAS BEEN FILLED AND THE WORK VECTOR INITIALIZED AS:"
250 PRINT "W1$=":FOR I=1 TO 3:HEXPRINT W1$ (I) :NEXT I:STOP
260 REM         CREATE THE LOCATOR ARRAY L$ WITH MAT MERGE
270 M=1
280 INIT (00) L$()
290 MAT MERGE M$ () TO W1$ () ,W2$ () ,L$ () :REM **********************
300 PRINT "MAT MERGE HAS BEEN EXECUTED":STOP
310 IF L$ (1) =HEX (0000)  THEN 570:REM   EXIT WHEN DONE
320 S=1
330 REM         DISPLAY THE CONTENTS OF L$ AND W1$
340 PRINT HEX (03) :PRINT "THE LOCATOR ARRAY L$=":FOR I=1 TO 4:HEXPRINT L$ (I)
:NEXT I:STOP
350 PRINT HEX (03) :PRINT "WORK VECTOR FROM MAT MERGE IS NOW" FOR I=1 TO 3
```

Program Listing (Continued)

```
HEXPRINT W1$ (I) :NEXT I:STOP
360 REM   MOVE THE MERGE ARRAY TO THE OUTPUT BUFFER O$ WITH MAT MOVE
370 PRINT HEX(03): PRINT "COUNTERS AND THE OUTPUT BUFFER ARE:"
380 N=4:PRINT "M=",M,"S=",S,"N=",N
390 MAT MOVE M$ () ,L$ (S) ,N TO O$ (M)
400 PRINT "N=",N,"ELTS  WERE MOVED"
410 REM   PUT DATA INTO OUTPUT ARRAY IF OUTPUT BUFFER FULL
420 M=M+N:REM   NO. OF ELTS. (ELEMENTS) PLACED IN OUTPUT BUFFER
430 R=R+1:IF M<=4 THEN 520
440 M=1:P=1:Q=4
450 FOR I=P TO Q:J$ (I) =O$ (I) :NEXT I
460 MAT COPY O$ () TO K$ () <(R-1)*20+1,20>
470 P=P+N:Q=Q+N:S=N+1:PRINT "P=",P,"Q=",Q,"S=",S
480 IF S<5 THEN 380:REM   RETURN TO MOVE IF MORE DATA TO MOVE
490 REM          DISPLAY THE CONTENTS OF O$
500 PRINT "OUTPUT ARRAY BUFFER"
510 FOR I=1 TO 4:PRINT J$ (I) ,:HEXPRINT O$ (I) ,:NEXT I:STOP
520 REM   CHECK TERMINATION OF MERGE
530 T=VAL (W1$ (3))  :PRINT "T=",T:STOP :IF T=0 THEN 280
540 IF L$ (1)<>HEX (0000)  THEN 280
550 PRINT HEX (03)
560 REM              DISPLAY THE MERGED ARRAY
570 PRINT "FINAL MERGED ARRAY"
580 FOR I=1 TO 4:J$ (I) =O$ (I) :NEXT I
590 FOR I=1 TO 8:PRINT K$ (I) :NEXT I
600 FOR I=1TO 4:PRINT J$ (I) ,:HEXPRINT O$ (I) ,:NEXT I:STOP
610 PRINT " ":PRINT "L$=":FOR I=1 TO 4:HEXPRINT L$ (I) :NEXT I:STOP
620 PRINT " ":PRINT "W1$=":FOR I=1 TO 3:HEXPRINT W1$ (I) :NEXT I
630 DATA "JAKE","JOHN","KARL","KATHY","JANE","JILL","KING","KITTY"
640 END
```

Program Printout

```
JAKE              JOHN             KARL             KATHY
JANE              JILL             KING             KITTY
THIS IS THE ARRAY I$#1
JAKE              JOHN             KARL             KATHY
JANE              JILL             KING             KITTY

THIS IS THE ARRAY I$#2
HEX OF THE MERGE ARRAY M$,OUTPUT OF MAT COPY
4A414B4520
4A4F484E20
4B41524020
4B41544859
4A414E4520
4A494C4C20
4B494E4720
4B49545459
```

Program Printout (Continued)

```
THE MERGE ARRAY HAS BEEN FILLED AND THE WORK VECTOR INITIALIZED
AS:
W1$=
01
01
01
MAT MERGE HAS BEEN EXECUTED
THE LOCATOR ARRAY L$=
0101
0201
0202
0102
WORK VECTOR FROM MAT MERGE IS NOW:
03
03
00
COUNTERS AND THE OUTPUT BUFFER ARE:
M=       1         S=        1
N=       4
N=       4         ELTS. WERE MOVED
P=       5         Q=        8
S=       5
OUTPUT ARRAY BUFFER
JAKE     4A414B4520
JANE     4A414E4520
JILL     4A494C4C20
JOHN     4A4F484E20
T=       0
MAT MERGE HAS BEEN EXECUTED
THE LOCATOR ARRAY L$=
0103
0104
0000
0000
WORK VECTOR FROM MAT MERGE IS NOW:
FF
03
01
COUNTERS AND THE OUTPUT BUFFER ARE:
M=       1       S=          1
N=       4
N=       2       ELTS. WERE MOVED
T=       1
MAT MERGE HAS BEEN EXECUTED
THE LOCATOR ARRAY L$=
0203
0204
0000
0000
WORK VECTOR FROM MAT MERGE IS NOW:
```

Program Printout (Continued)

```
FF
FF
02
COUNTERS AND THE OUTPUT BUFFER ARE:
M=                 3                S=                1
N=                 4
N=                 2                ELTS. WERE MOVED
P=                 3                Q=                6
S=                 3
M=                 1                S=                3
N=                 4
N=                 0                ELTS. WERE MOVED
T=                 2
MAT MERGE HAS BEEN EXECUTED
FINAL MERGED ARRAY

JAKE
JANE
JILL
JOHN
KARL             4B41524C20
KATHY            4B41544859
KING             4B494E4720
KITTY            4B49545459

L$=
0000
0000
0000
0000

W1$=
FF
FF
02
```

CHAPTER 2 SORT STATEMENTS

Chapter 2 contains a discussion of syntax and examples using the Sort statements. As with the other verbs and statements of the BASIC language, syntax is critically important when using the Sort statements. In addition, several of the Sort statements have array dimensioning requirements which must be followed. If the rules and restrictions presented in this manual are not followed exactly, programs using the Sort statements will not work.

In the general form provided for each MAT statement, arguments in brackets [ ] are optional. Stacked items in braces { } are alternatives; one must occur. Uppercase words and punctuation (commas, parentheses, etc.) represent actual characters in the statement which must occur exactly as shown. Lowercase words represent parameters which the programmer can specify.

No array to be used with the Sort statements can contain more than 4096 elements. The locator-array must always have elements of length=2.

NOTE:

Any data set containing numeric values must be changed to Sort format (alphanumeric) with MAT CONVERT before being used with the MAT SORT or MAT MERGE statement.

Do not use MAT REDIM on an array to reduce its total number of bytes and then execute a Sort statement operation on the array. MAT REDIM may be used with an array subsequently operated on by a Sort statement only if the total number of bytes in all elements after the MAT REDIM equals the total number before the MAT REDIM.

MAT CONVERT

General Form:

```
MAT CONVERT numeric-array TO alpha-array [(s,n)]
```

Where: (s,n) are expressions designating a field within each element of the alpha-array used to store the converted value. If (s,n) are omitted, the entire array is used.

s specifies the starting position of the field ($0 < s \leq$ bytes in the array).

n specifies the number of bytes in the field ($0 < n \leq$ bytes in the array - $s+1$).

The alpha-array must have at least as many elements as the numeric-array.

Purpose:

MAT CONVERT converts numeric-array-elements to alpha-array-elements which can be ordered in proper numeric sequence with MAT SORT. The format of data passed through the MAT CONVERT statement is called sort format.

Values in sort format can be stored in a specified field of each alpha-array element by using the field limiting expressions (s,n). The number of bytes specified in each field must be at least two. If more than eight bytes are specified, the value is truncated at the right, two digits per byte. For example, the statement:

```
MAT CONVERT  N() TO A$() (3,5)
```

specifies that the numeric values from the array N() are to be stored in the third through seventh bytes of each element of alpha-array A$(). If less than eight bytes are specified, the field is padded with zeroes.

Examples of valid syntax:

```
MAT CONVERT A() TO A$() (6,8)
MAT CONVERT N() TO A$()
```

Example 5: A Program Using MAT CONVERT

In the routine below, the numeric-array N() contains four numeric elements which are passed to A$() and B$() with MAT CONVERT statements and printed with the appropriate HEXPRINT statements. The field expressions in line 90 specify that each converted element value from N() is to be placed in the corresponding element of B$(), starting with the second character of each element of B$().

```
10 DIM N (4) ,A$ (2,2) 3,B$ (4) 8
20 N (1) =123: N (2) =-456: N (3) = .123:N (4) =0
30 PRINT "NUMERIC ARRAY,N  :"
40 FOR I= 1 TO 4:PRINT , N (I) : NEXT I
50 MAT CONVERT N() TO A$()
60 PRINT "ALPHA ARRAY, A$():"
70 PRINT ,:HEXPRINT A$ (1,1) :PRINT ,:HEXPRINT A$ (1,2)
80 PRINT ,:HEXPRINT A$ (2,1) :PRINT ,:HEXPRINT A$ (2,2)
90 MAT CONVERT N() TO B$() (2,3)
100 PRINT "ALPHA ARRAY, B$():"
110 FOR I= 1 TO 4:PRINT ,:HEXPRINT B$ (I) : NEXT I
```

The converted values are stored in both A$() and B$() with an exponent and three digits of the mantissa. In the printed output, hex(20) represents the space character.

Output from Program Using MAT CONVERT

```
NUMERIC ARRAY,N():
                  123
                 -456
                  .123
                  0
ALPHA ARRAY, A$():
                 902123
                 097543
                 898123
                 800000
ALPHA ARRAY, B$():
                 2090212320202020
                 2009754320202020
                 2089812320202020
                 2080000020202020

                     FIELD(2,3)=elements 2, 3 and 4 only
```

If it is necessary to reconvert values which have been passed through a MAT CONVERT statement back into standard internal numeric format, the Wang "Unconvert" utility should be used. A listing of this utility appears below.

Program Listing of the "Unconvert" Routine

```
1000 REM CONVERT BACK FROM SORT FORMAT TO NUMERIC
1010 REM ENTRY Z$ = NUMBER IN SORT FORMAT
1020 REM        Z1 = LENGTH OF ALPHA VARIABLE
1030 REM RETURN Z = NUMBER
1040 DIM Z$8,Z1$1,Z2$2,Z3$1
1050 DEFFN'200 (Z$,Z1)
1060 Z1$,Z2$ = Z$
1070 IF Z2$ = HEX (8000) THEN 1420 : REM ZERO EXIT
1080 Z$ = STR (Z$,2)  :REM MOVE UP INTO PACKED FORMAT
1090 AND (STR (Z$,1,1), OF)    :REM REMOVE EXTRA BITS
1100 REM EXTRACT THE EXPONENT
1110 Z3$ = STR (Z2$,2)
1120 AND (Z3$,FO)   : REM SECOND EXP. DIGIT
1130 AND (Z2$,OF)   : REM FIRST EXPT. DIGIT
1140 OR (Z3$,Z2$)   : REM EXP. REVERSED
1150 ROTATE (Z3$,4) : REM EXPONENT
1160 REM     NOW DETERMINE THE SIGN
1170 AND (Z1$,FO)  : REM SIGN BITS
1180 IF Z1$ = HEX (90)  THEN 1350  : REM EXP. +; MANTISSA +
1190 IF Z1$ = HEX (10)  THEN 1280  : REM EXP. -; MANTISSA -
1200 REM COMPLEMENT THE EXPONENT
1210 ADD (Z3$,66)
1220 XOR (Z3$,FF)
1230 IF Z1$ = HEX (80) THEN 1340  : REM EXP. -; MANTISSA +
1240 REM Z1$ = HEX (00)                EXP. +; MANTISSA -
1250 Z1$ = HEX (10)
1260 GOTO 1300 : REM COMPLEMENT MANTISSA
1270 REM Z1$ = HEX (10)
1280 Z1$ = HEX (90)
1290 REM COMPLEMENT MANTISSA
1300 ADD (Z$,66)
1310 XOR (Z$,FF)
1320 AND (STR(Z$,1,1),OF)
1330 REM INSERT SIGN
1340 OR (Z$,Z1$)
1350 STR (Z$,8) = Z3$   : REM INSERT EXPONENT
1360 REM ZERO UNUSED BYTES
1370 IF Z1>7 THEN 1390
1380 INIT (00) STR (Z$,Z1,8-Z1)   : REM ZERO UNUSED BYTES
1390 UNPACK (+#.#############     )Z$ TO Z
1400 RETURN
1410 REM ZERO EXIT
1420 Z = 0
1430 RETURN
```

MAT COPY

General Form:

MAT COPY [-] source-alpha-array [<s,n>] TO [-] output-alpha-array [<s,n>]

Where: (s,n) are expressions designating a field within either array. If (s,n) are omitted, the entire array is used.

s specifies the starting position of the field ($0 < s \leq$ bytes in the array).

n specifies the number of bytes in the field ($0 < n \leq$ bytes in the array - s+1).

Each array is treated as a contiguous string of bytes, ignoring element boundaries.

Purpose:

MAT COPY transfers data byte-by-byte from the source-alpha-array to the output-alpha-array. An array is treated as one contiguous character string, i.e., element boundaries are ignored. MAT COPY can be used to construct new arrays from old ones, combine or divide array elements, move data within arrays, and using the inverse parameter (-), store data in reverse order within arrays.

A portion of an array can be specified by using the field limiting expressions (s,n). "s" is the position of the first byte to be used in the array. The position is found by counting bytes across the first row from left-to-right, then across the second row, etc., until the desired byte is encountered. "n" specifies the number of bytes to be used. Data are moved until the output-array is filled. If the amount of data to be moved is insufficient to fill the specified portion of the output-array, the remainder of the array is filled with spaces.

If a minus sign (-) precedes the input-array name (or its specified portion), data are taken from this array in reverse order and stored left justified. The first byte moved is the last byte specified, and so on. If a minus sign (-) precedes the output-array, data are stored in the output-array in reverse order right justified, i.e., the first byte is stored in the last byte specified, etc.

Examples of valid syntax:

```
MAT COPY B$() TO C$()
MAT COPY - A$() TO B$()  <3,27>
MAT COPY A$() < X*Y, 100/X> TO -Q$() <10,20>
```

Example 6: MAT COPY

In this example, A$() has been dimensioned as a five-element array in which each element contains two bytes. B$() is dimensioned as a seven-element array in which each element contains two bytes. A$() contains the characters A to K, and these data are copied to B$(). As statement 30 is changed, the result is changed as shown below.

```
10 DIM A$(5)2, B$(7)2
20 FOR I=1 TO 5 : READ A$(I) : NEXT I
30 MAT COPY A$() TO B$()
40 DATA "AB", "CD", "EF", "GH", "JK"
50 FOR I = 1 TO 5 :  PRINT STR (A$ (I), 1); : NEXT I
60 PRINT
70 FOR I = 1 TO 5:  PRINT STR (B$ (I), 1); : NEXT I
80 PRINT
```

Output from MAT COPY Example

A$() =

A B	C D	E F	G H	J K

30 MAT COPY A$() TO B$()

B$() =

A B	C D	E F	G H	J K	space	space

30 MAT COPY A$() TO - B$()

B$() =

space	space	K J	H G	F E	D C	B A

30 MAT COPY-A$() TO B$()

B$() =

K J	H G	F E	D C	B A	space	space

30 MAT COPY - A$() TO - B$()

B$() =

space	space	A B	C D	E F	G H	J K

MAT MERGE

General Form:

```
MAT MERGE merge-alpha-array TO work-vector-1, work-vector-2,
          locator-array
```

Where: The merge-array must be a two-dimensional alpha-array with a maximum of 254 rows or columns; the number of rows (n) must be >1. The elements in the merge-array must already be sorted.

Work-vector-1 must be one-dimensional and possess at least one more element than the merge-array has rows. Each element must have length = 1 and be initialized to HEX(01) before initial use of the MAT MERGE statement.

Work-vector-2 must be one-dimensional and possess at least as many elements as the merge-array has rows. Each element must have length = 2. Although work-vector-2 is not used by the user's program, it must be defined for system use.

The locator-array must be one-dimensional and should possess at least as many elements as a single row of the merge-array. (It should have as many elements as possible because the larger the locator-array, the faster the merge.) Each element of the locator-array must have length = 2.

Each row of the merge-array must be presorted.

Purpose:

MAT MERGE enables the programmer to quickly combine the elements of several sorted data files into a single file by creating a locator-array of their subscripts in ascending order.

Each row of the merge-array represents a set of sorted data elements (one datum per element) to be merged and essentially constitutes a "merge buffer" for a given input file. The merge operation occurs in a series of passes where in each pass, the program fills the locator-array with elements from the merge-array, empties the locator-array to the output buffer, refills any emptied rows of the merge-array, and resets the pointers in work-vector-1. As each pass occurs, the data in each of the rows of the merge-array are scanned and compared. The subscripts corresponding to these data are then stored in the locator-array in order of increasing data value. The location pointer of work-array-1 is then updated to the next element in each row which is to be merged into the locator-array. When the locator-array is filled, the contents are written at the output device. MAT COPY replenishes empty rows in the merge-array. MAT MOVE moves the merged data from the locator-array to an output-array. These processes must be repeated until all elements in the merge-array have been exhausted. The merge operation halts when one of the following conditions occur.

1. The locator-array is filled with subscripts.

2. A row of the merge-array has been completely used (emptied of elements).

In the last element in work-vector-1, a condition code is returned which specifies why the merge pass terminated. The code is as follows:

00 - locator-array full
01 to FF - row number of emptied merge-array row

Each element of work-vector-1 receives the subscript of the next element in the row which will be compared with the elements in the other rows. All elements of work-vector-1 must be initialized to hex(01) (using INIT) before initial execution of MAT MERGE. Once a merge-array element is selected, its subscript is placed in the locator-array, and the work-vector-1 pointer increments to the next value in the row to be compared. When a row has been completely scanned, hex(FF) is placed as the pointer for that row in work-vector-1. The last element of work-vector-1 receives a condition code during MAT MERGE which indicates why the merge pass was halted. (If a row in the merge-array was exhausted of elements, the code is 01 to FF, designating the row number of the emptied row. If the locator-array was filled, the code is 00). The user's program must then refill that row of the merge-array with data, reinitialize the respective subscripts for that row of the merge-array in work-vector-1 to hex(01) and move the contents of the locator-array to the output buffer. If there are insufficient data to fill the row, the data must be right-justified in the row, and the starting subscript in the work-vector set appropriately. If no data remain to be placed in an empty row of the merge-array, hex(FF) is left in the corresponding element of work-vector-1 indicating that MAT MERGE is to ignore that row during successive merge passes. When MAT MERGE determines that all rows are to be ignored (hex (FF) is the pointer in all rows), hex(0000) is returned to the next element of the locator-array. The contents of the locator-array are again moved to the output buffer and execution of the merge phase is complete.

Examples of valid syntax:

```
10 DIM A$(10,50)10,W1$(11)1,W2$(10)2,S$(50)2
20 MAT MERGE A$() TO W1$(), W2$(), S$()
```

Example 7. A Three-Row Merge

In the following example, three rows of a merge-array A$ are merged using MAT MERGE. Each row may be regarded as a merge buffer. Prior to the merge, the data in each row must be sorted as in the following example where each row in the array is already in alphabetical order.

col:	1	2	3	4	5	6	
row 1	A	D	H	I	L	P	←——merge buffer 1
A$()= 2	B	C	F	J	M	Q	←——merge buffer 2
3	E	G	K	N	O	R	←——merge buffer 3

Work-vector-1 contains the pointers to the starting or current element in each row to be scanned for the merge pass. Each element of work-vector-1 is initially set (using INIT) to hex(01). The first three elements of work vector-1 each correspond to a row of the merge-array, as follows:

W1$()=	01	01	01	00
	row 1 pointer	row 2 pointer	row 3 pointer	condition code

Given the arrays A$() and W1$() as described above, the following statements will produce the merge results described in the following example.

```
10 DIM A$(3,6)1, W1$(4)1, W2$(3)2, L$(12)2

20 MAT MERGE A$() TO W1$(), W2$(), L$()
```

The subscript locator-array L$() is filled with the subscripts of the twelve lowest-order elements from the three merge-array rows. If the locator-array has not been filled by the execution of MAT MERGE (this occurs when a merge-array row is exhausted before the locator-array is full), the element following the last valid subscript contains hex(0000). During the first merge pass, twelve subscripts are placed in the locator-array from the merge-array as follows:

L$()=	0101	0201	0202	0102	0301	0203	0301	0103	0104	0204	0303	0105
	row 1 col 1	row 2 col 1	row 2 col 2	row 1 col 2	row 3 col 1							
elements	(A)	(B)	(C)	(D)	(E)	etc. of A$()						

Elements from all three rows in the merge-array are compared. MAT MERGE continues to extract subscripts and place them in ascending order in the locator-array until all the elements in a row have been exhausted or, as in this case, the locator-array is filled. The data which is in the locator-array must then be moved to an output-array using MAT MOVE.

The first merge pass terminates. Work-vector-1 is updated to indicate which element in each row is to be taken in the next merge pass. The first merge pass halted because the locator-array was full; thus the last element of W1$() is set to hex zero (00). W1$() now contains:

W1$() =	06	05	04	00
	row 1 elt 6	row 2 elt 5	row 3 elt 4	indicates locator-array full.

(elt = element in above row)

The second merge pass now occurs. Four elements are placed in the locator-array, L$(), before this merge pass terminates. L$() now contains:

L$() =

0205	0304	0305	0106	0000
(M)	(N)	(O)	(P)	

The second merge pass halted because row 1 of the merge-array was exhausted. W1$ will thus contain:

W1$() =

FF	06	06	01

If there is more data to be merged, row 1 of the merge-array must be refilled for re-execution of the MAT MERGE statement and the current element position for row 1 in work-vector-1, W1$(1), must be reset to hex(01). Since there are no more data to fill the row, however, the current element position for the row will remain hex(FF). When MAT MERGE is again executed, the row will be ignored.

The third merge pass will also terminate because of the emptying of a merge-array row (row 2). During this pass the locator-array will receive another element which is then moved to the output buffer. L$() and W1$() will thus contain:

L$() =

0206	0000
(Q)	

W1$() =

FF	FF	06	02

The fourth merge pass will now occur. L$() will receive one element during this merge pass. L$() and W1$() will thus contain:

L$() =

0306	0000
(R)	

W1$() =

FF	FF	FF	03

The fourth merge pass also terminates because a row of the merge-array was emptied (row 3). This pass also terminates the merge phase, however because all elements of the merge-array have been exhausted. The last record is moved from the locator-array to the output buffer and execution of the merge phase terminates.

MAT MOVE

General Form:

MAT MOVE has two general forms:

```
MAT MOVE source-alpha-array [(s,n)], starting-locator-array-element (r[,c])
         [,m] TO starting-output-alpha-array-element (r[,c])
```

<u>and</u>

```
MAT MOVE source-numeric-array, starting locator-array-element (r[,c])
         [,m] TO output-numeric-array-element (r[,c])
```

Where: (s,n) are expressions designating a field within each element of the source-alpha-array.

s specifies the starting position of the field (0< s≤ bytes in the array).

n specifies the number of bytes in the field (0< n≤ bytes in the array -s+1).

m is a numeric scalar variable specifying the maximum number of elements to be moved (0< m≤ 32767).

Purpose:

MAT MOVE transfers data element-by-element from one array to another in the order specified by the subscript locator-array. Source and output-arrays can either be alphanumeric or numeric; however, both arrays must be the same type.

Data is transferred into sequential bytes in the output-alpha-array, row by row. If the move-array and receiver-array are alphanumeric, data may be transferred between specified fields of each element. MAT MOVE continues to transfer data until:

1. an element whose value is (hex (0000)) is found in the locator-array,

2. the end of the locator-array is reached,

3. m elements have been moved, or

4. the output-array has been filled.

The locator-array row/column subscripts point to the data in the source array which are to be moved to the output-array. The first subscript moved is the element specified as the starting-locator-array-element. The first element in the output-array to receive data is specified as the starting-output-array-element. Data are moved sequentially, row-by-row, from that point on. If the field expressions (s,n) are given, only data in the specified field of each element of the source-array are moved. For example, the statement

```
MAT MOVE A$() (2,3), L$(1) TO B$(1)
```

specifies that only three bytes (the second, third and fourth) from each element of A$() are to be moved. If elements in the source-array contain fewer bytes than the elements in the output array, values are padded with spaces at the right. If elements in the source-array contain more bytes than the output-array-elements, they are truncated on the right.

When MAT MOVE has finished moving data, a count of the number of elements moved is returned to the variable m, if m has been specified.

Examples of valid syntax:

```
DIM A$(10,50)5,L$(50)2,A1$(100)5,A2$(50)5,A(10,50)
MAT MOVE A(), L$(1) TO A1(1)
MAT MOVE A$(),L$(10), G TO A2$(25)
MAT MOVE A$() (3,2), L$(1) TO A2$(1)
```

Example 8: MAT MOVE

Given an array of data called A$() and an associated locator-array called L$(), the following statements move the elements of A$() to the output array B$() in the order specified by the locator-array L$().

A$() =

row/col	1	2	3	4
1	F	A	G	I
2	H	E	J	D
3	C	L	B	K

L$() =

0102 (A)	0303 (B)	0301 (C)	0204 (D)
0202 (E)	0101 (F)	0103 (G)	0201 (H)
0104 (I)	0203 (J)	0304 (K)	0302 (L)

The locator-array provides the order in which data from the source-array is to be moved. L$(1,1) = 0102 means "move the element in row 1, column 2 of A$() first", L$(1,2) = 0303 means "move the element in row 3, column 3 next", etc. Subscripts in the locator-array appear in hexadecimal notation; letters in parentheses are the corresponding elements from A$(). This example assumes that L$(1,1) was specified as the starting-locator-array-element.

When the following statements are executed

```
10 DIM A$(3,4)1,L$(3,4)2,B$(3,4)1
20 MAT MOVE A$(),L$(1,1) TO B$(1,1)
```

the array B$() is created as:

B$() =

A	B	C	D
E	F	G	H
I	J	K	L

This is the output-array after execution of MAT MOVE, assuming that B$(1,1) was specified as the starting-output-array-element.

MAT SEARCH

General Form:

MAT SEARCH search-alpha-array-designator [<s,n>] , $\left\{\begin{array}{c} <> \\ > \\ < \\ <= \\ >= \\ = \end{array}\right\}$ alpha variable TO

location-array-designator [STEP expression]

Where: (s,n) are expressions defining a field within the search-array.

s specifies the starting position determined by counting bytes left-to-right across the first row, left-to-right across the next row, and so on, ignoring element boundaries. (0 < s ≤ bytes in the search-array).

n is the number of bytes to be searched. (0<n ≤ bytes in the search-array -s+1).

The STEP expression specifies that only substrings starting at every ith byte (where i = value of the STEP expression) are to be checked. The STEP expression must be 0 < expression ≤ 255.

The location-array can have a minimum of one element. The length of each element must be two bytes.

Purpose:

MAT SEARCH scans the search-array for substrings that exactly or partially satisfy some specified relation and stores the location of each such substring in the location-array. The locations are stored as two-byte binary values in the order in which they are found, giving the location of the substring from the beginning of the search-array (or specified portion of the search-array). A portion of the search-array can be scanned by using the field limiting expressions (s,n). If these expressions are omitted, the entire array is scanned, starting with the first element. The search-array is treated as a single contiguous character string, i.e., element boundaries are ignored. The length of the substrings scanned in the search-array equals the length of the value of the alpha-variable with which they are compared.

NOTE:

If it is necessary to check for trailing spaces (hex(20)) or find an exact match, use the STR() function to specify the exact number of characters to check, e.g., MAT SEARCH A$(), = STR(Z$,1,5) TO B$() specifies a search for five-character substrings in A$() which exactly equal the first five characters of Z$, including any trailing spaces. The STR() function ensures that trailing spaces are included in the value of Z$.

If the location-array is too small to accept all positions of the substrings satisfying the given relation, the search terminates when the location-array is full. If there is space remaining in the location-array after the search is complete, the next element in the location-array is filled with zero (hex(0000)).

If the STEP parameter is not used, a step of one byte is implied. The search-array is treated as a contiguous byte string. MAT SEARCH starts at the first character in the search-array and checks to see if the character string starting with that character satisfies the given relation. If so, the location of the string is stored as the first element of the location-array. Then the substring starting at the second character is checked, and so on. If the STEP parameter is used, the position of the next substring to be checked is the starting position of the last substring checked plus the STEP expression. The search terminates when the number of characters remaining to be checked in the search-array is less than the length of the value of the variable being compared.

Consider a data file A$() containing the names of the 50 states blocked such that each 14 bytes represents a state name (STEP 14). The Search key is B$, the location-array designator is C$().

CASE #1 - The exact match B$ = "MISSISSIPPI"
MAT SEARCH A$(), = STR (B$,1) TO C$() STEP 14
Returns the location of MISSISSIPPI in the file.

CASE #2 - The partial match B$ = "MISS"
MAT SEARCH A$(), = B$ TO C$() STEP 14
Returns the location of MISSISSIPPI and MISSOURI in the file.

CASE #3 - Any match B$ = "ISS"
MAT SEARCH A$(), = B$ TO C$()
Returns the positions of the first and second "ISS"
in MISSISSIPPI and the position of MISSOURI.

NOTE:

The binary values in the location-array produced by MAT SEARCH are byte locations, not element subscripts. The first byte in the array area to be searched is byte 0001, the second byte is 0002, the third byte is 0003, etc. The MAT SEARCH location-array cannot be used by MAT MOVE, unless these byte locations are first converted to subscripts.

Examples of valid syntax:

```
MAT SEARCH A$(), = B$ TO L$()
MAT SEARCH A$(), < Z$ TO L1$()
MAT SEARCH A$(), < R$ TO S1$() STEP 4
MAT SEARCH A$(), <S,N>, > STR(Q$,3,5) TO S$()
```

Example 9: MAT SEARCH

Given array A$() as follows:

row/col =		1	2	3	4
	1	B	A	C	D
A$() =	2	C	B	E	A
	3	A	F	A	E

and a location-array B$() dimensioned B$(12)2, when the statements

```
20  INIT (FF) B$() : O$="A"
30  MAT SEARCH A$(), =O$ TO B$()
```

are executed, the array B$() receives the locations of the character "A" in array A$(). These locations are counted from the first element specified, across each row left to right and stored as binary values. A binary zero is stored after the last location stored in B$(). B$() will contain:

row/col		1	2	3	4	5	6	 12
B$() =	1	0002	0008	0009	000B	0000	FFFF	FFFF

If desired, the two-byte binary values in the location-array can be converted to decimal form and assigned to a numeric variable by a statement such as:

```
D = 256*VAL(B$(J)) + VAL(STR(B$(J),2))
```

where:

B$()	is the location-array.
J	is the subscript of the location-array element to be converted.
D	is the numeric equivalent of the two-byte binary location.

Or, if desired, the binary location-array elements of B$() could be converted to a set of subscripts which define the position of the search strings in the original search-array. This would allow the data which was searched for to be moved with MAT MOVE to another output file with a routine such as:

```
10 DIM A1$(12)4: REM LOCATOR-ARRAY FOR MAT MOVE
20 INIT ("0") A1$()
30 INPUT "STARTING ELEMENT IS", S
40 INPUT "ENDING ELEMENT IS", N
50 I = 1
60 IF B$(I) = HEX (0000) THEN 110
70 CONVERT INT ((S-2+VAL(STR(B$(I),2)))/4)+1 TO STR(A1$(I),1,2),(##)
80 CONVERT VAL(STR(B$(I), 2))+S-1-(INT((S-2+VAL(STR(B$(I),2)))/4*4))
TO STR(A1$(I), 3, 2),(##)
90 I = I+1
100 GO TO 60
110 MAT PRINT A1$; :REM A1$() contains the subscript pointers to the
locations of the search strings in the original search-array


:RUN
0102
0204
0301
0303
0000
  .
  .
  .
0000
```

MAT SORT

General Form:

MAT SORT sort-array TO work-array, locator-array

Where: The work-array and the locator-array have at least as many elements as the sort-array. The elements in these arrays must be of length=2.

The sort-array cannot contain more than 4096 elements.

Purpose:

MAT SORT takes the elements of the sort-array and creates an output locator array of their subscripts arranged according to the ascending order of the elements in the sort-array. Subscripts are stored in the locator-array as two-byte values; the first byte is the row subscript, the second byte is the column subscript. If the locator-array contains more elements than the sort-array (n), then the nth + 1 element of the locator-array contains zero (hex(0000)).

MAT MOVE uses the locator-array to create a new data array in sorted order.

NOTE:

The subscripts of sort-array elements of identical value are stored contiguously in the locator-array, but they are not necessarily ordered as in the sort-array.

Example of valid syntax:

```
MAT SORT A$() TO W$(), L$()
```

If G$() =

row/col	1	2	3	4
1	C	A	E	I
2	L	J	G	B
3	H	D	K	E

The following statements take a sort-array of twelve elements, G$(), and create a subscript array G1$().

```
10 DIM G$(3,4)1, W$(3,4)2, G1$(3,4)2
20 MAT SORT G$() TO W$(), G1$()
```

G1$()=

0102	0204	0101	0302
0304	0103	0203	0301
0104	0202	0303	0201

Thus, the lowest value is in element (1,2), i.e., the letter A in row 1, column 2; the second lowest value is in element (2,4), i.e., the letter B in row 2, column 4, etc.

NOTE:

Every byte of an alphanumeric array in the System 2200 is set to all blank characters (HEX(20)) when initially defined unless explicitly set to some other character with an INIT statement. Since HEX(20) is lower in the collating (sorting) sequence than other standard characters, HEX(20)'s will always float to the top (beginning) of any sorted array. It is therefore good practice to initialize arrays to be sorted with an INIT(FF) statement. This will ensure that the unused elements of any array will sink to the bottom (end) of the sorted array.

Example 10: Using MAT SORT and MAT MOVE

In the following short program, an input or sort array, I$(), is processed by MAT SORT which fills the locator-array L$() containing the subscript pointers. Then the locator-array is used by MAT MOVE to pick off the elements of the sort-array in sorted order and place them in the sorted output-array, O$() .

```
10 DIM I$(10)1, W$(10)2, L$(12)2, O$(10)1
20 FOR I=1 TO 10 : INPUT I$(I) : NEXT I
30 MAT SORT I$() TO W$(), L$()
40 FOR I=1 TO 12 : HEXPRINT L$(I) : NEXT I
50 M=10
60 PRINT
70 MAT MOVE I$(), L$(1), M TO O$(1)
80 FOR I=1 TO 10 : PRINT O$(I) : NEXT I
```

Input up to ten single letter array elements (not in alphabetical order). The locator-array is printed in hex-codes; the output-array is printed as usual.

```
:RUN (EXEC)
? S, D, E, A, Z, X, Y, G, B, J (EXEC)
040109010201030108010A01010106010701050100002020
ABCDEGJSXYZ
```

Figure 1-8. Output from Example 1

APPENDIX A: ERROR CODES

Code: =1

Error: MISSING NUMERIC ARRAY DESIGNATOR

Cause: A numeric array designator (e.g., N()) was expected.

Action: Correct the statement in error.

Example:

```
100 MAT CONVERT A$() TO N()
                   ↑ERR=1

100 MAT CONVERT N() TO A$()             (Possible Correction)
```

Code: =2

Error: ARRAY TOO LARGE

Cause: The specified array contains too many elements. For example, the number of elements cannot exceed 4096.

Action: Correct the program.

Example:

```
10 DIM A$(100,50)2, B$(100,50)2, W$(100,50)2
.
.
.
100 MAT SORT A$() TO S$(), B$()
                              ↑ERR=2

10 DIM A$(100,40)2, B$(100,40)2, W$(100,40)2
                                        (Possible Correction)
```

Code: =3

Error: ILLEGAL DIMENSIONS

Cause: The defined dimensions or element length of the array are illegal.

Action: Dimension the array properly.

Example:

```
10 DIM A$(63), B$(63)1, W$(63)2
.
.
.
100 MAT SORT A$() TO W$(), B$()
                              ↑ERR=3

10 DIM A$(63), B$(63)2, W$(63)2         (Possible Correction)
```

APPENDIX B: MAT SORT/MAT MERGE TIMING

The timing characteristics of the algorithms chosen to implement MAT SORT should be considered when planning sorting applications. The speed with which data are sorted depends upon the number of passes made through the data, the amount of work done on each pass to reduce the data to sorted order and the characteristics of the data. It is a feature of the MAT SORT algorithm that judicious choice of array size and dimensions can produce marked improvements in execution time for sorting operations. The program in Appendix E is provided for this purpose.

Execution time for MAT SORT is generally proportional to the expression:

$$n(\log_2 n)^2$$

where n is the number of elements in the array to be sorted. The worst results occur when n is a power of two (i.e., $n=2^x$); and the best results occur when $=(2^x+1)$. (See Table and Figure B-1). As n increases, substantial improvements in overall sorting time can be achieved by separating the elements to be sorted into groups, sorting within each group and then using MAT MERGE to merge the groups.

While each additional row of data added to the merge array causes a linear increase in the time required for sorting when using both MAT SORT and MAT MERGE, (see Table and Figure B.2), this bookkeeping time reduces the potentially exponential effect of the necessary across-row comparisons. Therefore, the user is advised to choose a constant row size to correspond to one of the optimal cases in the MAT SORT routine. A good choice might be 129 elements in a row since 128 is the largest power of two less than 255, the maximum row length.

While these characteristics of MAT SORT and MAT MERGE can be of value to the experienced user, it is most likely that setup time for large arrays will greatly exceed actual sorting time. To aid in minimizing setup time, MAT MOVE and MAT COPY should be used for their ability to move large volumes of data rapidly within memory. The greatest total sort speeds will be exhibited by BASIC programs which utilize all the Sort ROM commands effectively.

The tables and graphs illustrating MAT SORT/MAT MERGE timing for several array sizes have been empirically determined; dots are actual times as given in the tables.

Table B-1. MAT SORT Timing

Number of Elements	Time (in seconds)
100	0.66
128*	1.20
129	0.98
200	1.68
256*	3.80
258	2.30
300	2.95
400	4.60
500	7.20
510	10.20
512*	11.00
513	6.20
600	7.80
800	12.00
1000	18.00
1024*	30.90
1025	15.50
2000	43.00

*Powers of 2

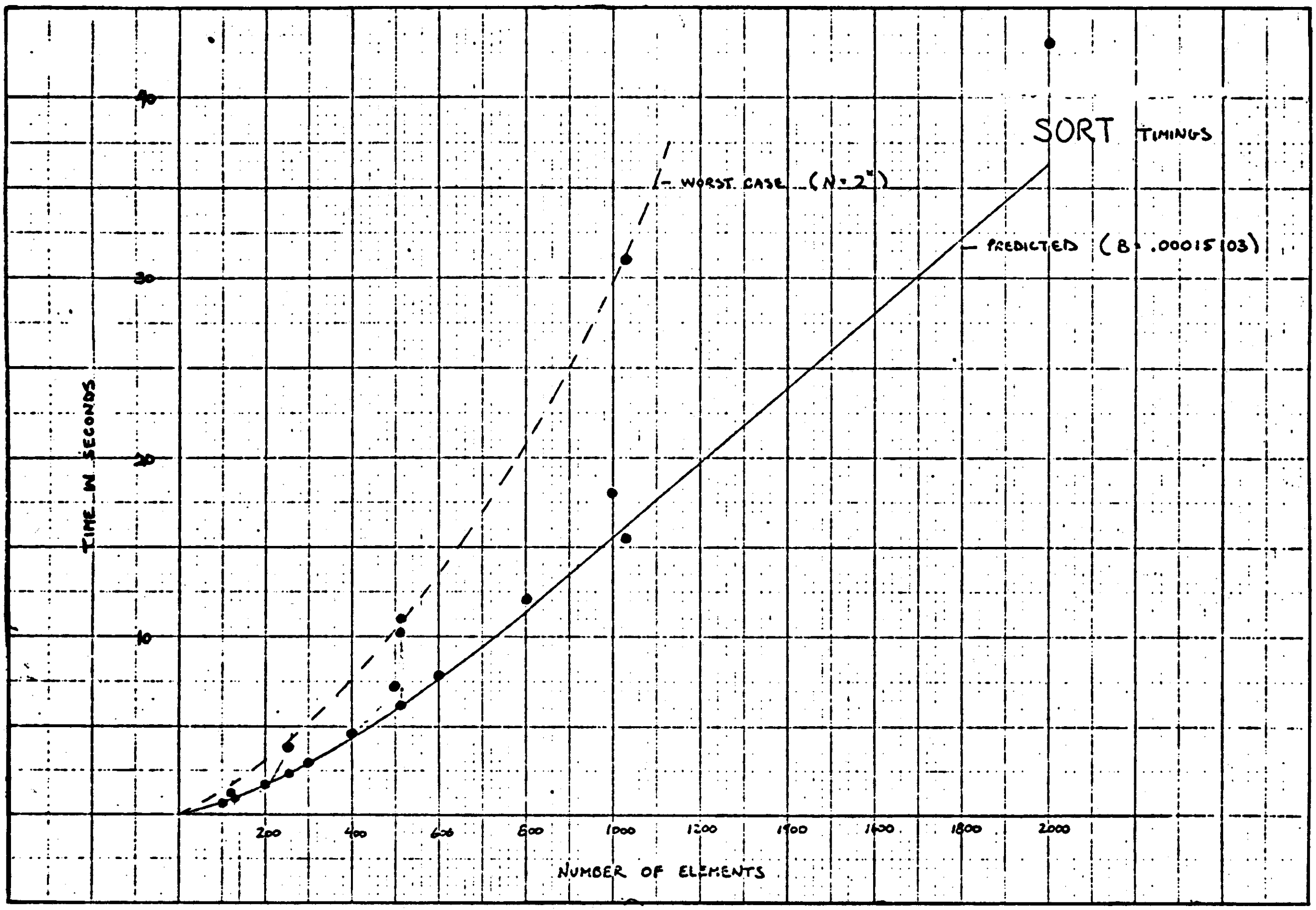


Figure B-1. MAT SORT Timing

Table B-2. MAT SORT/MAT MERGE Timing

Number of Elements	Time (in seconds)
500 (5 x 100)	4.6
645 (5 x 129)	6.0
774 (6 x 129)	7.0
903 (7 x 129)	9.0
1032 (8 x 129)	10.0
1161 (9 x 129)	11.0
1290 (10 x 129)	13.0
2000 (20 x 100)	22.0

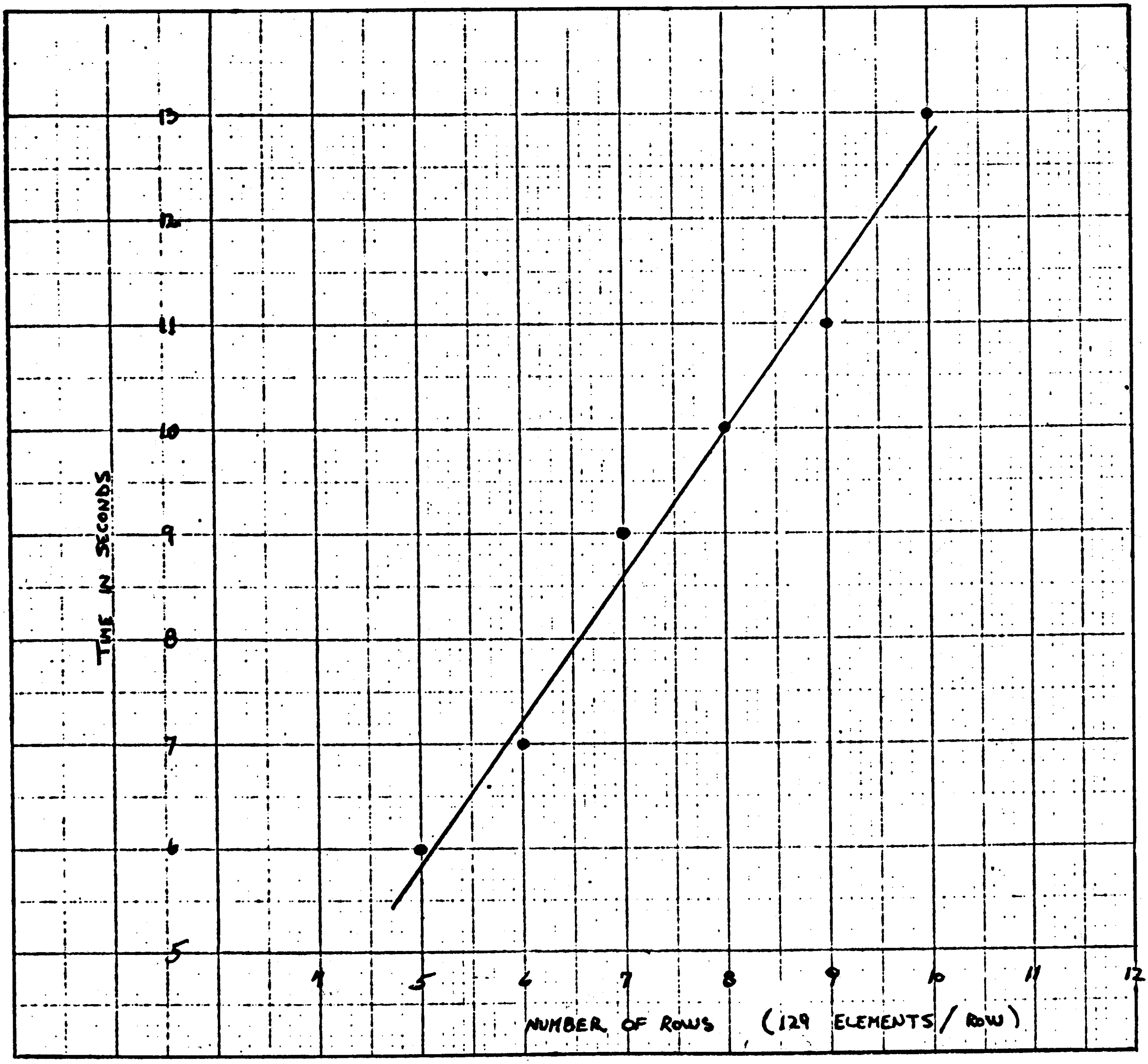


Figure B-2. MAT SORT/MAT MERGE Timing

APPENDIX C: SOME USEFUL DEFINITIONS

array: a number of related elements arranged in a specified order. Arrays can be one- or two-dimensional. A one-dimensional array has one row and is sometimes called a vector; a two-dimensional array has both rows and columns.

byte: a sequence of adjacent binary digits (bits) operated on as a unit; one byte contains eight bits.

field: a set of bytes within a record (or array) specified for use by a particular category of data.

key: one or more characters or fields within a record that are used to identify it or control its use.

locate mode: a means of accessing data by pointing to its location instead of moving it.

merge: to combine items from two or more similarly ordered sets into a single set that is arranged in the same order.

pointer: an address or other indication of location.

record: a collection of related items of data treated as a unit (for example, one line of an invoice may form a record; one item in the line such as quantity is a field).

sort: to segregate items into groups according to some rule or rules.

string: A subfile of records stored in sorted order.

subscript: an identifying character or number written below and to the right of another character. In Sort statements, the locator-arrays contain subscripts of alpha-array elements.

vector: a one-dimensional array.

work array: an intermediate array used for temporary storage of data between phases.

APPENDIX D: ASCII, HEX, Binary and Decimal (VAL) Equivalents Table

Wang HEX	Wang CRT Character Set	Wang VAL	Wang HEX	Wang CRT Set	Wang VAL	Wang HEX	Wang CRT Set	Wang VAL	Wang HEX	Wang CRT Set	Wang VAL
00	NUL	0	40	@	64	80		128	C0		192
01	cursór home	1	41	A	65	81		129	C1		193
02		2	42	B	66	82		130	C2		194
03	Clear screen	3	43	C	67	83		131	C3		195
04		4	44	D	68	84		132	C4		196
05		5	45	E	69	85		133	C5		197
06		6	46	F	70	86		134	C6		198
07	alarm	7	47	G	71	87		135	C7		199
08	backup cursor	8	48	H	72	88		136	C8		200
09	HT	9	49	I	73	89		137	C9		201
0A	LF	10	4A	J	74	8A		138	CA		202
0B	VT	11	4B	K	75	8B		139	CB	K	203
0C	FF	12	4C	L	76	8C		140	CC		204
0D	CR	13	4D	M	77	8D		141	CD	E	205
0E	SO	14	4E	N	78	8E		142	CE		206
0F	SI	15	4F	O	79	8F	K	143	CF	Y	207
10		16	50	P	80	90		144	D0		208
11	X-ON	17	51	Q	81	91	E	145	D1	W	209
12		18	52	R	82	92		146	D2		210
13	X-OFF	19	53	S	83	93	Y	147	D3	O	211
14		20	54	T	84	94		148	D4		212
15		21	55	U	85	95		149	D5	R	213
16		22	56	V	86	96	W	150	D6		214
17		23	57	W	87	97		151	D7	D	215
18		24	58	X	88	98	O	152	D8		216
19	cl. tab	25	59	Y	89	99		153	D9	S	217
1A	set tab	26	5A	Z	90	9A	R	154	DA		218
1B		27	5B	[	91	9B		155	DB		219
1C		28	5C	\	92	9C	D	156	DC		220
1D		29	5D	]	93	9D		157	DD		221
1E	ç	30	5E	↑	94	9E	S	158	DE		222
1F	°(degree)	31	5F	←	95	9F		159	DF		223
20	space	32	60	(prime)	96	A0		160	E0		224
21	!	33	61	a	97	A1		161	E1		225
22	"	34	62	b	98	A2		162	E2		226
23	#	35	63	c	99	A3		163	E3		227
24	$	36	64	d	100	A4		164	E4		228
25	%	37	65	e	101	A5		165	E5		229
26	&	38	66	f	102	A6		166	E6		230
27	' (apos.)	39	67	g	103	A7		167	E7		231
28	(	40	68	h	104	A8		168	E8		232
29	)	41	69	i	105	A9		169	E9		233
2A	*	42	6A	j	106	AA		170	EA		234
2B	+	43	6B	k	107	AB		171	EB		235
2C	, (comma)	44	6C	l	108	AC		172	EC		236
2D	- (minus)	45	6D	m	109	AD		173	ED		237
2E	.	46	6E	n	110	AE		174	EE		238
2F	/	47	6F	o	111	AF		175	EF		239
30	0	48	70	p	112	B0		176	F0		240
31	1	49	71	q	113	B1		177	F1		241
32	2	50	72	r	114	B2		178	F2		242
33	3	51	73	s	115	B3		179	F3		243
34	4	52	74	t	116	B4		180	F4		244
35	5	53	75	u	117	B5		181	F5		245
36	6	54	76	v	118	B6		182	F6		246
37	7	55	77	w	119	B7		183	F7		247
38	8	56	78	x	120	B8		184	F8		248
39	9	57	79	y	121	B9		185	F9		249
3A	:	58	7A	z	122	BA		186	FA		250
3B	;	59	7B	{	123	BB		187	FB		251
3C	<	60	7C	¦	124	BC		188	FC		252
3D	=	61	7D	}	125	BD		189	FD		253
3E	>	62	7E	~	126	BE		190	FE		254
3F	?	63	7F	▮	127	BF		191	FF		255

APPENDIX E: APPROXIMATING THE PARAMETERS FOR SORT/MERGE

This program is designed to approximate the optimal sizes of the Sort, Merge, Locator, Work and Output arrays for a given memory capacity, input file and Sort key length.

```
10 REM PROG DETERMINES SIZES OF COMPONENTS FOR SORT/MERGE-2200T
20 REM CRT/PRINTER INDICATOR
30 DIM C$1
40 REM ROUND UP ROUTINE
50 DEFFNU(X)=INT(X+.9999999999999)
60 SELECT PRINT 005(64)
70 PRINT HEX(03);"PROGRAM TO DETERMINE SIZES OF COMPONENTS FOR SORT/MERGE (2200T)";HEX(0A)
80 INPUT "MEMORY SIZE (K)",M1
90 INPUT "MAXIMUM NUMBER OF RECORDS",N
100 INPUT "NUMBER OF RECORDS PER SECTOR",B
110 REM MAXIMUM SIZE SORT KEY = 60 IF UNBLOCKED
120 K0=60
130 IF B=1 THEN 160
140 REM MAXIMUM SIZE SORT KEY = 59 IF BLOCKED
150 K0=59
160 PRINT "LENGTH OF SORT KEY (MAX";K0;")";
170 INPUT K
180 IF K<1 THEN 160
190 IF K>K0 THEN 160
200 IF K<>INT(K) THEN 160
210 REM AVAILABLE MEMORY FOR EITHER SORT OR MERGE PHASE
220 M2=M1*1024-698-1400
230 REM IF ONE RECORD PER SECTOR, ADDRESS POINTER IS 2 BYTES
240 K1=2
250 REM 124 TWO BYTE ADDRESSES PER SECTOR IN TAG FILE
260 K5=124
270 IF B=1 THEN 330
280 REM IF BLOCKED RECORDS, ADDRESS POINTER IS 3 BYTES
290 K1=3
300 REM 80 THREE BYTE ADDRESSES PER SECTOR IN TAG FILE
310 K5=80
320 REM TOTAL KEY/ADDRESS LENGTH
330 K2=K+K1
340 REM NUMBER OF KEYS+ADDRESSES PER SECTOR, COLS IN SORT ARRAY
350 K3=INT(253/(K2+1))
360 REM MEMORY REQUIRED FOR SORT OUTPUT / MERGE INPUT BUFFER
370 K4=K2*K3+7
380 REM MEMORY REQUIRED FOR MERGE OUTPUT BUFFER
390 K6=K1*K5+7
400 REM ELEMENT SIZE OF DISK RECORD FOR MERGE OUTPUT (TAG) FILE
410 K7=K1*K5/4
420 REM NUMBER OF DIVISIONS OF ORIGINAL FILE REQUIRED
430 D=FNU(N/FNU((M2-K4)/(K2+4)))
440 REM MAXIMUM RECORDS PER DIVISION
450 N2=K3*FNU(FNU(N/D)/K3)
460 REM TOTAL SECTORS REQUIRED FOR SORT OUTPUT FILE
470 F2=N2/K3+2
480 REM ABSOLUTE MAXIMUM NUMBER OF RECORDS
490 N1=N2*D
500 REM TOTAL SECTORS REQUIRED FOR MAXIMUM INPUT FILE
510 F1=FNU(N1/B+2)
520 REM ROWS IN SORT/WORK/LOCATOR ARRAYS
530 R=N2/K3
540 REM MEMORY REQUIRED BY SORT ARRAY
550 S1=R*K3*K2+7
560 REM MEMORY REQUIRED BY WORK ARRAY
570 S2=R*K3*2+7
580 REM MEMORY REQUIRED BY SORT LOCATOR ARRAY
590 S3=R*K3*2+7
600 REM TOTAL MEMORY REQUIRED BY SORT/WORK/LOCATOR/OUTPUT
610 M4=S1+S2+S3+K4+M4
620 REM COLUMNS REQUIRED FOR MERGE & MERGE LOCATOR ARRAY
630 X=INT(N2/D)
640 IF X<=255 THEN 660
650 X=255
660 C=K3*INT(X/K3)
670 REM MEMORY REQUIRED BY MERGE ARRAY
680 T1=D*C*K2+7
690 REM MEMORY REQUIRED BY WORK ONE ARRAY
700 T2=D+1+7
710 REM MEMORY REQUIRED BY WORK TWO ARRAY
720 T3=D*2+7
730 REM MEMORY REQUIRED BY MERGE LOCATOR ARRAY
740 T4=D*C*2+7
750 REM TOT MEMORY REQUIRED BY MERGE/WORK1/WORK2/LOCATOR/OUTPUT
760 M5=K4+T1+T2+T3+T4+K6
770 REM MAXIMUM RECORDS IN TAG FILE
780 N3=K5*FNU(N1/K5)
790 REM TOTAL SECTORS REQUIRED FOR MERGE OUTPUT (TAG) FILE
800 F3=N3/K5+2
810 INPUT "ENTER 1 FOR CRT, 2 FOR PRINTER, 3 TO STOP",Z
820 IF Z=1 THEN 860
830 IF Z=2 THEN 890
840 IF Z=3 THEN 1450
850 GOTO 810
860 SELECT PRINT 005(64)
870 C$=HEX(03)
880 GOTO 910
890 SELECT PRINT 215(64)
900 C$=HEX(0A)
```

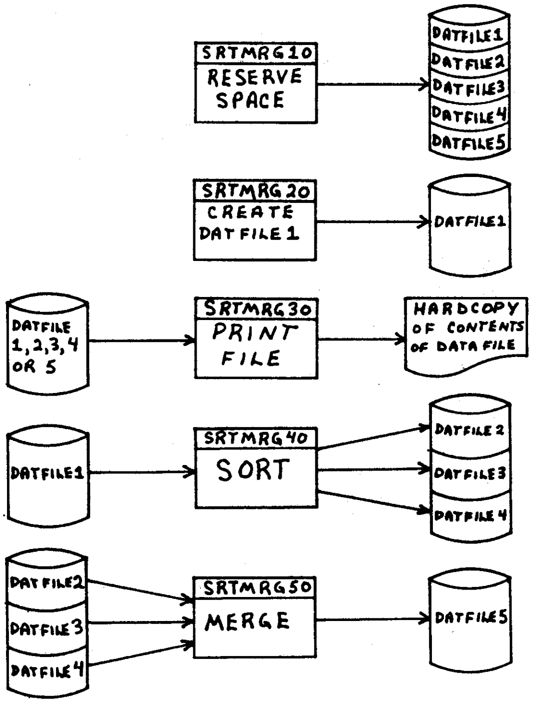
```
910 PRINT C$
920 PRINT HEX(0E);"*****  MEMORY  *****"
930 PRINT M1;"K SYSTEM";TAB(32);1024*M1
940 PRINT "SYSTEM OVERHEAD (2200T)";TAB(32);-698
950 PRINT "SORT OR MERGE PROGRAM";TAB(32);-1400
960 PRINT "NET MEMORY AVAILABLE";TAB(32);M2
970 PRINT HEX(0A)
980 PRINT HEX(0E);"*****  INPUT FILE  *****"
990 PRINT "MAX RECORDS (";N;")";TAB(32);N1
1000 PRINT "NO. REC PER SECTOR";TAB(32);B
1010 PRINT "TOT NO. SECTORS";TAB(32);F1
1020 PRINT HEX(0A)
1030 PRINT HEX(0E);"*****  SORT KEY  *****"
1040 PRINT "SORT KEY LENGTH";TAB(32);K
1050 PRINT "ADDRESS POINTER";TAB(32);K1
1060 PRINT "TOT KEY/ADD LENGTH";TAB(32);K2
1070 IF C$=HEX(0A) THEN 1090
1080 INPUT "KEY EXEC TO RESUME",X$
1090 PRINT C$
1100 PRINT HEX(0E);"*****  SORT PHASE  *****"
1110 PRINT "SORT ARRAY          S$(";R;",";K3;")";K2;TAB(40);S1
1120 PRINT "WORK ARRAY          W$(";R;",";K3;") 2";TAB(40);S2
1130 PRINT "LOCATOR ARRAY       L$(";R;",";K3;") 2";TAB(40);S3
1140 PRINT "OUTPUT BUFFER       O$(";K3;")";K2;TAB(40);K4
1150 PRINT "TOTAL MEMORY";TAB(40);M4
1160 PRINT HEX(0A)
1170 PRINT "OUTPUT FILES","NUMBER NEEDED",D
1180 PRINT ,"MAX. RECORDS",N2
1190 PRINT ,"RECORDS/SECTOR",K3
1200 PRINT ,"TOT SECT/FILE",F2
1210 IF C$=HEX(0A) THEN 1240
1220 PRINT HEX(0A)
1230 INPUT "KEY EXEC TO RESUME",X$
1240 PRINT C$
1250 PRINT HEX(0E);"*****  MERGE PHASE  *****"
1260 PRINT "INPUT BUFFER        I$(";K3;")";K2;TAB(40);K4
1270 PRINT "MERGE ARRAY         M$(";D;",";C;")";K2;TAB(40);T1
1280 PRINT "WORK ARRAY 1        W1$(";D+1;") 1";TAB(40);T2
1290 PRINT "WORK ARRAY 2        W2$(";D;") 2";TAB(40);T3
1300 PRINT "LOCATOR ARRAY       L$(";D;",";C;") 2";TAB(40);T4
1310 PRINT "OUTPUT BUFFER       O$(";K5;")";K1;TAB(40);K6
1320 PRINT "TOTAL";TAB(40);M5
1330 PRINT HEX(0A)
1340 PRINT "OUTPUT FILE","MAXIMUM RECORDS",N3
1350 PRINT ,"RECORDS/SECTOR",K5
1360 PRINT ,"TOTAL SECTORS",F3
1370 PRINT ,"NOTE:  MERGE OUTPUT BUFFER AND OUTPUT TAG FILE"
1380 PRINT ,"DIMENSIONED IN MEMORY  AS   O$(";K5;")";K1
1390 PRINT ,"RECORDS STORED ON DISK AS   O$( 4 )";K7
1400 IF C$=HEX(0A) THEN 1420
1410 INPUT "KEY EXEC TO RESUME",X$
1420 PRINT HEX(0C)
1430 PRINT C$
1440 GOTO 810
1450 STOP
```

```
*****    MEMORY    *****
 16 K SYSTEM                      16384
SYSTEM OVERHEAD (2200T)          -698
SORT OR MERGE PROGRAM            -1400
NET MEMORY AVAILABLE              14286

*****    INPUT FILE    *****
MAX RECORDS ( 1000 )             1008
NO. REC PER SECTOR               1
TOT NO. SECTORS                  1010

*****    SORT KEY    *****
SORT KEY LENGTH                  31
ADDRESS POINTER                  2
TOT KEY/ADD LENGTH               33

*****    SORT PHASE    *****
SORT ARRAY          S$( 48 , 7 ) 33      11095
WORK ARRAY          W$( 48 , 7 ) 2       679
LOCATOR ARRAY       L$( 48 , 7 ) 2       679
OUTPUT BUFFER       O$( 7 ) 33           238
TOTAL MEMORY                             12691

OUTPUT FILES    NUMBER NEEDED    3
                MAX. RECORDS     336
                RECORDS/SECTOR   7
                TOT SECT/FILE    50

*****    MERGE PHASE    *****
INPUT BUFFER        I$( 7 ) 33           238
MERGE ARRAY         M$( 3 , 112 ) 33     11095
WORK ARRAY 1        W1$( 4 ) 1           11
WORK ARRAY 2        W2$( 3 ) 2           13
LOCATOR ARRAY       L$( 3 , 112 ) 2      679
OUTPUT BUFFER       O$( 124 ) 2          255
TOTAL                                    12291

OUTPUT FILE     MAXIMUM RECORDS  1116
                RECORDS/SECTOR   124
                TOTAL SECTORS    11
                NOTE:  MERGE OUTPUT BUFFER AND OUTPUT TAG FILE
                DIMENSIONED IN MEMORY  AS   O$( 124 ) 2
                RECORDS STORED ON DISK AS   O$( 4 ) 62
```

APPENDIX F: EXTERNAL SORT/MERGE EXAMPLE

This program is designed to demonstrate a common external sorting technique.

```
-----------------------------------
I     SORT - MERGE    EXAMPLE     I
-----------------------------------


-----------------------------------
I     PROGRAM DESCRIPTIONS        I
-----------------------------------

SRTMRG10   -   RESERVE SPACE
RESERVES SPAVE FOR THE DATA FILES REQUIRED FOR THE SORT - MERGE
EXAMPLE.

SRTMRG20   -   CREATE DATFILE1
USES THE RANDOM NUMBER GENERATOR TO CREATE 90 SIX DIGIT NUMBERS
THEY ARE BLOCKED  10  PER SECTOR  AND  STORED  AS  ALPHANUMERIC
CHARACTERS,  I.E.  A$(10)6.   THEY  ARE  WRITTEN  TO  DATFILE1.

SRTMRG30   -   PRINT FILE
PRINTS THE CONTENTS OF THE DATA FILES USED BY THE  SORT - MERGE
EXAMPLE.       THE NAME OF THE FILE IS ENTERED BY THE OPERATOR.

SRTMRG40   -   SORT
READS IN  DATFILE1  IN THREE GROUPS, SORTS EACH GROUP AND WRITES
EACH GROUP TO SEPARATE FILES (DATFILE2, DATEFILE3, & DATFILE4).

SRTMRG50   -   MERGE
MERGES  THE THREE SORTED FILES (DATFILE2, DATFILE3, & DATFILE4)
INTO ONE FILE (DATFILE5).


-----------------------------------
I     DATA FILE DESCRIPTIONS      I
-----------------------------------

DATFILE1
THE ORIGINAL DATA IN RANDOM ORDER.  11 SECTORS RESERVED.   NINE
LOGICAL RECORDS OF 1 SECTOR EACH.  EACH LOGICAL RECORD CONTAINS
10   6-DIGIT  NUMBERS  STORED  AS  6  ALPHANUMERIC  CHARACTERS.

DATFILE2,  DATFILE3,  &  DATFILE4
EACH FILE CONTAINS ONE-THIRD OF THE  ORIGINAL  DATA  SORTED  IN
ASCENDING SEQUENCE.  5 SECTORS RESERVED FOR EACH FILE.  LOGICAL
RECORDS ARE THE SAME AS DATFILE1

DATFILE5
ORIGINAL  DATA  SORTED IN ASCENDING SEQUENCE.  SAME DESCRIPTION
AS DATFILE1.
```



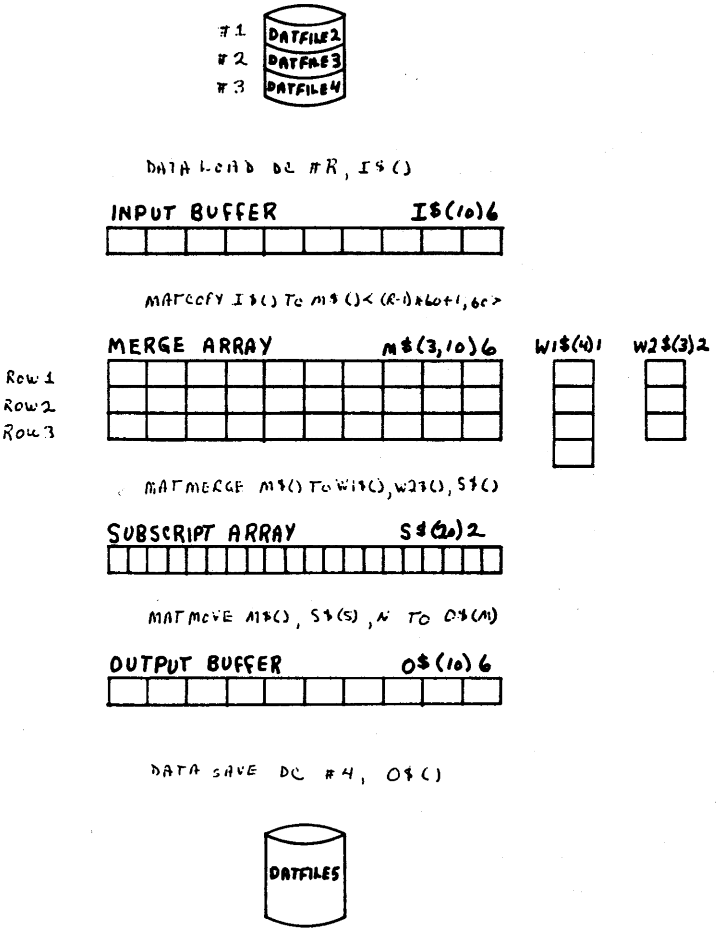
```
10 REM PROGRAM NAME = "SRTMRG10"
20 REM THIS PROGRAM RESERVES SPACE ON DISK FOR THE DATA FILES
30 REM FOR THE SORT-MERGE EXAMPLE
40 SELECT #1 310
50 SELECT PRINT 005(64)
60 PRINT HEX(03);"PROCESSING SRTMRG10"
70 PRINT "ALLOCATING SPACE FOR THE FIVE DATAFILES"
80 DATA SAVE DC OPEN T#1, 11, "DATFILE1"
90 DATA SAVE DC #1, END
100 DATA SAVE DC OPEN T#1,  5, "DATFILE2"
110 DATA SAVE DC #1, END
120 DATA SAVE DC OPEN T#1,  5, "DATFILE3"
130 DATA SAVE DC #1, END
140 DATA SAVE DC OPEN T#1,  5, "DATFILE4"
150 DATA SAVE DC #1, END
160 DATA SAVE DC OPEN T#1, 11, "DATFILE5"
170 DATA SAVE DC #1, END
180 DATA SAVE DC CLOSEALL
190 STOP "- SPACE ALLOCATED"
```

```
10 REM PROGRAM NAME = "SRTMRG20"
20 REM THIS PROGRAM CREATES THE DATA FOR "DATFILE1" VIA THE
30 REM RANDOM NUMBER GENERATOR
40 SELECT #1 310
50 SELECT PRINT 005(64)
60 PRINT HEX(03);"PROCESSING SRTMRG20"
70 PRINT "CREATING DATFILE1"
80 DIM A$(10)6
90 DATA LOAD DC OPEN T#1, "DATFILE1"
100 FOR I=1 TO 9
110 FOR J=1 TO 10
120 X=RND(X)
130 CONVERT X*1000000 TO A$(J), (######)
140 NEXT J
150 DATA SAVE DC #1, A$()
160 NEXT I
170 DATA SAVE DC #1, END
180 DATA SAVE DC CLOSE ALL
190 STOP "- DATFILE1 CREATED"
```

```
10 REM PROGRAM NAME = "SRTMRG30"
20 REM THIS PROGRAM PRINTS THE CONTENTS OF THE DATA FILES USED
30 REM IN THE SORT-MERGE EXAMPLE.  FILE NAME IS ENTERED.
40 SELECT #1 310
50 SELECT PRINT 005(64)
60 PRINT HEX(03);"PROCESSING SRTMRG30"
70 PRINT "PRINTS THE CONTENTS OF DATFILE1, DATFILE2, DATFILE3, DATFILE4,"
80 PRINT "OR DATFILE5"
90 DIM A$(10)6,F$8
100 INPUT "NAME OF FILE TO BE PRINTED",F$
110 DATA LOAD DC OPEN T#1, F$
120 SELECT PRINT 215(132)
130 PRINT HEX(0C0E);F$
140 PRINT HEX(0A0A)
150 N=0
160 DATA LOAD DC #1, A$()
170 IF END THEN 260
180 N=N+1
190 PRINT N;
200 FOR I=1 TO 10
210 PRINT TAB(8*I);A$(I);
220 NEXT I
230 PRINT
240 PRINT HEX(0A0A)
250 GOTO 160
260 DATA SAVE DC CLOSE ALL
270 PRINT HEX(0C)
280 SELECT PRINT 005(64)
290 STOP "- FILE PRINTED"
```

```
10 REM PROGRAM NAME = "SRTMRG40"
20 REM THIS PROGRAM READS IN "DATFILE1" AND CREATES 3 SORTED
30 REM SUBFILES - "DATFILE2", "DATFILE3", & "DATFILE4"
40 SELECT #1 310, #2 310
50 SELECT PRINT 005(64)
60 PRINT HEX(03);"PROCESSING SRTMRG40"
70 PRINT "CREATING SORTED SUBFILES DATFILE2, DATFILE3, DATFILE4 FROM"
80 PRINT "DATFILE1"
90 DIM A$(10)6,S$(3,10)6,W$(30)2,L$(30)2,F$8
100 F$="DATFILE1"
110 DATA LOAD DC OPEN T#1, F$ : REM OPEN INPUT FILE
120 SELECT PRINT 215 (132) : PRINT HEX(OC)
130 FOR I=1 TO 3 : REM DIVIDE FILE INTO THIRDS
140 PRINT HEX(OE);"GROUP #";I
150 FOR J=1 TO 3 : REM EACH THIRD CONTAINS 3 REC'S OF 10 NO'S
160 DATA LOAD DC #1,A$() : REM READ RECORD
170 GOSUB 390 : REM PRINT CONTENTS OF INPUT BUFFER
180 MAT COPY A$() TO S$()<1+60*(J-1),60> : REM TRANSFER INPUT BFFR TO SORT ARRAY
190 NEXT J
200 MAT SORT S$() TO W$(),L$() : REM ESTABLISH LOC ARRAY FOR MOVE
210 CONVERT I+1 TO STR(F$,8,1),(#) : REM "DATFILEx" WHERE x=I+1
220 DATA LOAD DC OPEN T#2, F$ : REM OPEN OUTPUT WORK FILE
230 PRINT HEX(0A0E);F$
240 FOR J=1 TO 21 STEP 10 : REM MOVE 10 AT A TIME TO OUTPUT BFFR AND SAVE
250 M=10 : REM MAXIMUM TO MOVE
260 MAT MOVE S$(),L$(J),M TO A$(1) : REM MOVE 10 TO OUTPUT BFFR
270 GOSUB 390 : REM PRINT CONTENTS OF OUTPUT BUFFER
280 DATA SAVE DC #2, A$() : REM WRITE RECORD
290 NEXT J : REM NEXT GROUP OF 10
300 DATA SAVE DC #2, END : REM WRITE EOF
310 DATA SAVE DC CLOSE#2 : REM CLOSE OUTPUT WORK FILE
320 PRINT HEX(0A0A0A)
330 NEXT I : REM NEXT THIRD
340 DATA SAVE DC CLOSE ALL : REM CLOSE ALL FILES
350 PRINT HEX(0C)
360 SELECT PRINT 005(64)
370 STOP "- SUBFILES CREATED"
380 REM PRINT CONTENTS OF A$() ON ONE LINE
390 FOR K=1 TO 10:PRINT TAB(10*(K-1));A$(K);:NEXT K:PRINT :RETURN
```

```
10 REM PROGRAM NAME = "SRTMRG50"
20 REM MERGES DATFILE2, DATFILE3, & DATFILE4 INTO DATFILES
30 SELECT PRINT 005(64):PRINT HEX(03);"PROCESSING SRTMRG50"
40 SELECT #1 310, #2 310, #3 310, #4 310, PRINT 215(132)
50 DIM M$(3,10)6,I$(10)6,O$(10)6,W1$(4)1,W2$(3)2,S$(20)2
60 REM OPEN INPUT WORK FILES
70 DATA LOAD DC OPEN T#1, "DATFILE2"
80 DATA LOAD DC OPEN T#2, "DATFILE3"
90 DATA LOAD DC OPEN T#3, "DATFILE4"
100 REM OPEN OUTPUT FILE
110 DATA LOAD DC OPEN T#4, "DATFILES"
120 PRINT HEX(OC)
130 GOTO 220
140 DEFFN' 40 (R) : REM SUBROUTINE TO FILL ROW "R"
150 PRINT "         FILL ROW";R;
160 DATA LOAD DC #R, I$() : REM READ RECORD FROM FILE R
170 IF END THEN 200 : REM IF EOF LEAVE ROW POINTER AT HEX(FF)
180 MAT COPY I$() TO M$()<60*(R-1)+1,60> : REM TRANSFER TO ROW R
190 W1$(R)=HEX(01) : REM RESET ROW POINTER TO COLUMN ONE
200 W1$(4)=HEX(00) : REM RESETS TERMINATION INDICATOR
210 RETURN
220 FOR T=1 TO 3 : REM FILL ALL THREE ROWS INITIALLY
230 GOSUB '40(T)
240 PRINT
250 NEXT T
260 M=1 : REM SET LOCATION IN OUTPUT BUFFER TO ONE
270 X=0 : REM SET # MERGE PASSES TO ZERO
280 INIT(00) S$() : REM RESET SUBSCRIPT ARRAY TO ZEROES
290 PRINT "          W1$() = ";:FOR I=1 TO 4:HEXPRINT W1$(I);:PRINT " ";:NEXT I:PRINT
300 MAT MERGE M$() TO W1$(), W2$(), S$() : REM OBTAIN SUBSCRIPTS FOR MOVE
310 X=X+1 : REM UPDATE # MERGE PASSES
320 PRINTUSING 570,X;:FOR I=1 TO 20:PRINT " ";:HEXPRINT S$(I);:NEXT I:PRINT
330 IF S$(1)=HEX(0000) THEN 520 : REM IF TRUE WE ARE DONE
340 S=1 : REM SET LOCATION IN SUBSCRIPT ARRAY TO ONE
350 N=10 : REM MAXIMUM TO MOVE IS TEN
360 MAT MOVE M$(), S$(S), N TO O$(M) : REM MOVE TO OUTPUT BUFFER
370 PRINTUSING 580,N;:FOR I=1 TO 10:PRINT TAB(7*I+22);O$(I);:NEXT I
380 M=M+N : REM UPDATE LOCATION IN OUTPUT BUFFER
390 IF M<=10 THEN 460 : REM IS OUTPUT BUFFER FULL?
400 DATA SAVE DC #4, O$() : REM IF SO WRITE RECORD
410 PRINT " -- WRITE RECORD"
420 INIT(20)O$() : REM RESET OUTPUT BUFFER TO EMPTY CONDITION
430 M=1 : REM RESET LOCATION IN OUTPUT BUFFER TO ONE
440 S=S+N : REM UPDATE LOCATION IN SUBSCRIPT ARRAY
450 IF S<=20 THEN 350 : REM HAVE WE MOVED ALL ELEMENTS PICKED THIS PASS?
460 PRINT
470 T=VAL(W1$(4)) : REM DETERMINE WHY MERGE TERMINATED
480 IF T=0 THEN 280 : REM TERM. BECAUSE SUBSCRIPT ARRAY FULL, BACK TO MERGE
490 PRINT "          W1$() = ";:FOR I=1 TO 4:HEXPRINT W1$(I);:PRINT " ";:NEXT I
500 GOSUB '40(T) : REM TERM. BECAUSE ROW R EXHAUSTED, REFILL
510 GOTO 280 : REM BACK TO MERGE
520 DATA SAVE DC #4, END : REM ALL ITEMS MERGED, WRITE EOF
530 DATA SAVE DC CLOSE ALL : REM CLOSE ALL FILES
540 PRINT HEX(0C)
550 SELECT PRINT 005(64)
560 STOP "- END OF MERGE"
570 %## MERGE S$()=
580 %          MOVED ## --- O$() =
DATFILE1

1      894597  227622  398691  391328  764136  619831  433854  700900  892217  280910

2      605643  029278  617768  367182  037928  559163  373715  314792  883153  485123

3      467036  773048  851732  790345  705707  971082  586873  658933  342659  579627

4      907383  056206  425670  918218  293593  511187  579624  907511  059016  598621

5      262524  748218  198796  309451  540922  289569  787202  785401  831193  302148

6      063249  786489  803526  374913  378784  584236  748981  219188  356444  631532

7      807871  267042  406918  017131  663520  203904  197885  401352  258292  926735

8      478261  751955  298449  183105  046164  489025  306815  369828  105687  945345

9      875897  650122  611897  176199  722724  494547  082241  038041  618271  383448


GROUP # 1
894597     227622     398691     391328     764136     619831     433854     700900     892217     280910
605643     029278     617768     367182     037928     559163     373715     314792     883153     485123
467036     773048     851732     790345     705707     971082     586873     658933     342659     579627

DATFILE2
029278     037928     227622     280910     314792     342659     367182     373715     391328     398691
433854     467036     485123     559163     579627     586873     605643     617768     619831     658933
700900     705707     764136     773048     790345     851732     883153     892217     894597     971082


GROUP # 2
907383     056206     425670     918218     293593     511187     579624     907511     059016     598621
262524     748218     198796     309451     540922     289569     787202     785401     831193     302148
063249     786489     803526     374913     378784     584236     748981     219188     356444     631532

DATFILE3
056206     059016     063249     198796     219188     262524     289569     293593     302148     309451
356444     374913     378784     425670     511187     540922     579624     584236     598621     631532
748218     748981     785401     786489     787202     803526     831193     907383     907511     918218


GROUP # 3
807871     267042     406918     017131     663520     203904     197885     401352     258292     926735
478261     751955     298449     183105     046164     489025     306815     369828     105687     945345
875897     650122     611897     176199     722724     494547     082241     038041     618271     383448

DATFILE4
017131     038041     046164     082241     105687     176199     183105     197885     203904     258292
267042     298449     306815     369828     383448     401352     406918     478261     489025     494547
611897     618271     650122     663520     722724     751955     807871     875897     926735     945345

DATFILE5

1      017131  029278  037928  038041  046164  056206  059016  063249  082241  105687

2      176199  183105  197885  198796  203904  219188  227622  258292  262524  267042

3      280910  289569  293593  298449  302148  306815  309451  314792  342659  356444

4      367182  369828  373715  374913  378784  383448  391328  398691  401352  406918

5      425670  433854  467036  478261  485123  489025  494547  511187  540922  559163

6      579624  579627  584236  586873  598621  605643  611897  617768  618271  619831

7      631532  650122  658933  663520  700900  705707  722724  748218  748981  751955

8      764136  773048  785401  786489  787202  790345  803526  807871  831193  851732

9      875897  883153  892217  894597  907383  907511  918218  926735  945345  971082
```



```
        FILL ROW 1
        FILL ROW 2
        FILL ROW 3
        W1$() = 01 01 01 00
1 MERGE S$()= 0301 0101 0102 0302 0303 0201 0202 0203 0304 0305 0306 0307 0308 0204 0309 0205 0103 030A 0000 0000
        MOVED 10 --- O$() = 017131 029278 037928 038041 046164 056206 059016 063249 082241 105687 -- WRITE RECORD
        MOVED  8 --- O$() = 176199 183105 197885 198796 203904 219188 227622 258292
        W1$() = 04 06 FF 03          FILL ROW 3          W1$() = 04 06 01 00
2 MERGE S$()= 0206 0301 0104 0207 0208 0302 0209 0303 020A 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
        MOVED  2 --- O$() = 176199 183105 197885 198796 203904 219188 227622 258292 262524 267042 -- WRITE RECORD
        MOVED  7 --- O$() = 280910 289569 293593 298449 302148 306815 309451
        W1$() = 05 FF 04 02          FILL ROW 2          W1$() = 05 01 04 00
3 MERGE S$()= 0105 0106 0201 0107 0304 0108 0202 0203 0305 0109 010A 0000 0000 0000 0000 0000 0000 0000 0000 0000
        MOVED  3 --- O$() = 280910 289569 293593 298449 302148 306815 309451 314792 342659 356444 -- WRITE RECORD
        MOVED  8 --- O$() = 367182 369828 373715 374913 378784 383448 391328 398691
        W1$() = FF 04 06 01          FILL ROW 1          W1$() = 01 04 06 00
4 MERGE S$()= 0306 0307 0204 0101 0102 0308 0103 0309 030A 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
        MOVED  2 --- O$() = 367182 369828 373715 374913 378784 383448 391328 398691 401352 406918 -- WRITE RECORD
        MOVED  7 --- O$() = 425670 433854 467036 478261 485123 489025 494547
        W1$() = 04 05 FF 03          FILL ROW 3          W1$() = 04 05 01 00
5 MERGE S$()= 0205 0206 0104 0207 0105 0208 0106 0209 0107 0301 0108 0302 0109 020A 0000 0000 0000 0000 0000 0000
        MOVED  3 --- O$() = 425670 433854 467036 478261 485123 489025 494547 511187 540922 559163 -- WRITE RECORD
        MOVED 10 --- O$() = 579624 579627 584236 586873 598621 605643 611897 617768 618271 619831 -- WRITE RECORD
        MOVED  1 --- O$() = 631532
        W1$() = 0A FF 03 02          FILL ROW 2          W1$() = 0A 01 03 00
6 MERGE S$()= 0303 010A 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
        MOVED  2 --- O$() = 631532 650122 658933
        W1$() = FF 01 04 01          FILL ROW 1          W1$() = 01 01 04 00
7 MERGE S$()= 0304 0101 0102 0305 0201 0202 0306 0103 0104 0203 0204 0205 0105 0206 0307 0207 0106 0308 0107 0108
        MOVED  7 --- O$() = 631532 650122 658933 663520 700900 705707 722724 748218 748981 751955 -- WRITE RECORD
        MOVED 10 --- O$() = 764136 773048 785401 786489 787202 790345 803526 807871 831193 851732 -- WRITE RECORD
        MOVED  3 --- O$() = 875897 883153 892217
        W1$() = 09 08 09 00
8 MERGE S$()= 0109 0208 0209 020A 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
        MOVED  4 --- O$() = 875897 883153 892217 894597 907383 907511 918218
        W1$() = 0A FF 09 02          FILL ROW 2          W1$() = 0A FF 09 00
9 MERGE S$()= 0309 030A 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
        MOVED  2 --- O$() = 875897 883153 892217 894597 907383 907511 918218 926735 945345
        W1$() = 0A FF FF 03          FILL ROW 3          W1$() = 0A FF FF 00
10 MERGE S$()= 010A 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
        MOVED  1 --- O$() = 875897 883153 892217 894597 907383 907511 918218 926735 945345 971082 -- WRITE RECORD
        MOVED  0 --- O$() =
        W1$() = FF FF FF 01          FILL ROW 1          W1$() = FF FF FF 00
11 MERGE S$()= 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
```

APPENDIX G: INTERNAL SORT/SEARCH EXAMPLES

These programs exemplify and compare various internal sorting and searching techniques.

```
10 REM PROGRAM TO ILLUSTRATE THE USE OF MAT SORT AND THE THREE
20 REM FORMS OF MAT SEARCH; EXACT, PARTIAL, & ANY CHAR STRING
30 DIM S1$(50)14,S2$(50)14,A1$(50)2,A2$(50)2,W$(50)2,L$(50)2,S$14
40 RESTORE : MAT READ S1$,A1$ : GOSUB '0 : STOP "- SELECT FUNCTION"
50 REM .......... FUNCTION LIST
60 DEFFN'0
70 PRINT HEX(03);"COMPARISON OF SORT ROM AND BASIC"
80 PRINT "THE DATA USED IS THE 50 STATES AND THEIR ABBREVIATIONS";HEX(0A)
90 PRINT "FN KEY     FUNCTION"
100 PRINT "------     --------------------------------"
110 PRINT "  '0       FUNCTION LIST"
120 PRINT "  '1       ORIGINAL DATA"
130 PRINT "  '2       SORT USING MAT SORT"
140 PRINT "  '3       SEARCH USING MAT SEARCH  -  EXACT ENTRY"
150 PRINT "  '4       SEARCH USING MAT SEARCH  -  PARTIAL ENTRY"
160 PRINT "  '5       SEARCH USING MAT SEARCH  -  ANY CHARACTER STRING"
170 RETURN
180 REM .......... ORIGINAL DATA
190 DEFFN'1
200 PRINT HEX(03);"ORIGINAL DATA"
210 FOR I=1 TO 50 : PRINT S1$(I), : NEXT I : RETURN
220 REM .......... SORT USING MAT SORT
230 DEFFN'2
240 PRINT HEX(03);"SORT WITH MAT SORT"
250 MAT SORT S1$() TO W$(),L$()
260 MAT MOVE S1$(), L$(1) TO S2$(1)
270 MAT MOVE A1$(), L$(1) TO A2$(1)
280 FOR I=1 TO 50 : PRINT S2$(I), : NEXT I : RETURN
290 REM .......... SEARCH USING MAT SEARCH - EXACT ENTRY
300 DEFFN'3
310 PRINT HEX(03);"SEARCH USING MAT SEARCH  -  EXACT ENTRY";HEX(0A)
320 S$=" "
330 INPUT "STATE (EXEC IF DONE)",S$
340 IF S$<>" " THEN 360
350 RETURN
360 MAT SEARCH S1$(),=STR(S$,1) TO L$() STEP 14
370 IF L$(1)>HEX(0000) THEN 400
380 PRINT "NOT ON FILE"
390 GOTO 320
400 X=256*VAL(L$(1))+VAL(STR(L$(1),2)) : REM CONVERT BINARY TO DECIMAL
410 Y=INT((X-1)/14)+1 : REM CONVERT CHARACTER TO ARRAY ELEMENT (1-50)
420 PRINT Y,S1$(Y),A1$(Y)
430 GOTO 320
440 REM .......... SEARCH USING MAT SEARCH  -  PARTIAL ENTRY
450 DEFFN'4
460 PRINT HEX(03);"SEARCH USING MAT SEARCH  -  PARTIAL ENTRY"
470 PRINT
480 S$=" "
490 INPUT "STATE (EXEC IF DONE)",S$
500 IF S$<>" " THEN 520
510 RETURN
520 MAT SEARCH S1$(),=S$ TO L$() STEP 14
530 IF L$(1)>HEX(0000) THEN 560
540 PRINT "NOT ON FILE"
550 GOTO 470
560 I=1
570 IF L$(I)=HEX(0000) THEN 470
580 X=256*VAL(L$(I))+VAL(STR(L$(I),2)) : REM CONVERT BINARY TO DECIMAL
590 Y=INT((X-1)/14)+1 : REM CONVERT CHARACTER TO ARRAY ELEMENT (1-50)
600 PRINT Y,S1$(Y),A1$(Y)
610 I=I+1
620 IF I<=50 THEN 570
630 GOTO 470
640 REM .......... SEARCH USING MAT SEARCH  -  ANY CHAR STRING
650 DEFFN'5
660 PRINT HEX(03);"SEARCH USING MAT SEARCH  -  ANY CHARACTER STRING"
670 PRINT
680 S$=" "
690 INPUT "CHARACTER STRING (EXEC IF DONE)",S$
700 IF S$<>" " THEN 720
710 RETURN
720 MAT SEARCH S1$(),=S$ TO L$()
730 IF L$(1)>HEX(0000) THEN 760
740 PRINT "NOT ON FILE"
750 GOTO 670
760 I=1
770 IF L$(I)=HEX(0000) THEN 670
780 X=256*VAL(L$(I))+VAL(STR(L$(I),2)) : REM CONVERT BINARY TO DECIMAL
790 Y=INT((X-1)/14)+1 : REM CONVERT CHARACTER TO ARRAY ELEMENT (1-50)
800 Z=X-14*(Y-1) : REM CHARACTER WITHIN ARRAY ELEMENT (1-14)
810 PRINT "ELEMENT";Y,"CHAR LOC";Z,S1$(Y),A1$(Y)
820 I=I+1
830 IF I<=50 THEN 770
840 GOTO 670
850 DATA "HAWAII","TENNESSEE","VIRGINIA","NEW YORK","MAINE","SOUTH DAKOTA","SOUTH CAR
OLINA","NEW HAMPSHIRE","IOWA","WYOMING","INDIANA","MISSISSIPPI","DELAWARE","ALABAMA",
"GEORGIA","NEW MEXICO"
860 DATA "MASSACHUSETTS","KENTUCKY","UTAH","WEST VIRGINIA","CALIFORNIA","MINNESOTA","
OREGON","OKLAHOMA","WISCONSIN","WASHINGTON","ALASKA","PENNSYLVANIA","MICHIGAN","FLORI
DA","NORTH CAROLINA"
870 DATA "MARYLAND","NEVADA","IDAHO","VERMONT","OHIO","LOUSIANA","TEXAS","NEBRASKA","
ILLINOIS","ARKANSAS","MONTANA","MISSOURI","ARIZONA","RHODE ISLAND","KANSAS","CONNECTI
CUT","NEW JERSEY"
880 DATA "COLORADO","NORTH DAKOTA"
890 DATA "HI","TN","VA","NY","ME","SD","SC","NH","IA","WY","IN","MS","DE","AL","GA","
NM","MA","KY","UT","WV","CA","MN","OR","OK","WI","WA","AK","PA","MI","FL","NC","MD","
NV","ID","VT","OH","LA"
900 DATA "TX","NE","IL","AR","MT","MO","AZ","RI","KS","CT","NJ","CO","ND"
```

```
     ORIGINAL DATA                         SORTED DATA
 ---------------------------         ---------------------------
ITEM     STATE         ABBR         ITEM     STATE         ABBR
----  --------------   ----         ----  --------------   ----
  1   HAWAII            HI            1   ALABAMA           AL
  2   TENNESSEE         TN            2   ALASKA            AK
  3   VIRGINIA          VA            3   ARIZONA           AZ
  4   NEW YORK          NY            4   ARKANSAS          AR
  5   MAINE             ME            5   CALIFORNIA        CA
  6   SOUTH DAKOTA      SD            6   COLORADO          CO
  7   SOUTH CAROLINA    SC            7   CONNECTICUT       CT
  8   NEW HAMPSHIRE     NH            8   DELAWARE          DE
  9   IOWA              IA            9   FLORIDA           FL
 10   WYOMING           WY           10   GEORGIA           GA
 11   INDIANA           IN           11   HAWAII            HI
 12   MISSISSIPPI       MS           12   IDAHO             ID
 13   DELAWARE          DE           13   ILLINOIS          IL
 14   ALABAMA           AL           14   INDIANA           IN
 15   GEORGIA           GA           15   IOWA              IA
 16   NEW MEXICO        NM           16   KANSAS            KS
 17   MASSACHUSETTS     MA           17   KENTUCKY          KY
 18   KENTUCKY          KY           18   LOUSIANA          LA
 19   UTAH              UT           19   MAINE             ME
 20   WEST VIRGINIA     WV           20   MARYLAND          MD
 21   CALIFORNIA        CA           21   MASSACHUSETTS     MA
 22   MINNESOTA         MN           22   MICHIGAN          MI
 23   OREGON            OR           23   MINNESOTA         MN
 24   OKLAHOMA          OK           24   MISSISSIPPI       MS
 25   WISCONSIN         WI           25   MISSOURI          MO
 26   WASHINGTON        WA           26   MONTANA           MT
 27   ALASKA            AK           27   NEBRASKA          NE
 28   PENNSYLVANIA      PA           28   NEVADA            NV
 29   MICHIGAN          MI           29   NEW HAMPSHIRE     NH
 30   FLORIDA           FL           30   NEW JERSEY        NJ
 31   NORTH CAROLINA    NC           31   NEW MEXICO        NM
 32   MARYLAND          MD           32   NEW YORK          NY
 33   NEVADA            NV           33   NORTH CAROLINA    NC
 34   IDAHO             ID           34   NORTH DAKOTA      ND
 35   VERMONT           VT           35   OHIO              OH
 36   OHIO              OH           36   OKLAHOMA          OK
 37   LOUSIANA          LA           37   OREGON            OR
 38   TEXAS             TX           38   PENNSYLVANIA      PA
 39   NEBRASKA          NE           39   RHODE ISLAND      RI
 40   ILLINOIS          IL           40   SOUTH CAROLINA    SC
 41   ARKANSAS          AR           41   SOUTH DAKOTA      SD
 42   MONTANA           MT           42   TENNESSEE         TN
 43   MISSOURI          MO           43   TEXAS             TX
 44   ARIZONA           AZ           44   UTAH              UT
 45   RHODE ISLAND      RI           45   VERMONT           VT
 46   KANSAS            KS           46   VIRGINIA          VA
 47   CONNECTICUT       CT           47   WASHINGTON        WA
 48   NEW JERSEY        NJ           48   WEST VIRGINIA     WV
 49   COLORADO          CO           49   WISCONSIN         WI
 50   NORTH DAKOTA      ND           50   WYOMING           WY
```

LOCATOR ARRAY

```
                                DECIMAL       HEXADECIMAL
                              -----------     -----------
ITEM       STATE              FROM    TO      FROM    TO
----   --------------         ----   ----     ----   ----
  1    HAWAII                   1     14      0001   000E
  2    TENNESSEE               15     28      000F   001C
  3    VIRGINIA                29     42      001D   002A
  4    NEW YORK                43     56      002B   0038
  5    MAINE                   57     70      0039   0046
  6    SOUTH DAKOTA            71     84      0047   0054
  7    SOUTH CAROLINA          85     98      0055   0062
  8    NEW HAMPSHIRE           99    112      0063   0070
  9    IOWA                   113    126      0071   007E
 10    WYOMING                127    140      007F   008C
 11    INDIANA                141    154      008D   009A
 12    MISSISSIPPI            155    168      009B   00A8
 13    DELAWARE               169    182      00A9   00B6
 14    ALABAMA                183    196      00B7   00C4
 15    GEORGIA                197    210      00C5   00D2
 16    NEW MEXICO             211    224      00D3   00E0
 17    MASSACHUSETTS          225    238      00E1   00EE
 18    KENTUCKY               239    252      00EF   00FC
 19    UTAH                   253    266      00FD   010A
 20    WEST VIRGINIA          267    280      010B   0118
 21    CALIFORNIA             281    294      0119   0126
 22    MINNESOTA              295    308      0127   0134
 23    OREGON                 309    322      0135   0142
 24    OKLAHOMA               323    336      0143   0150
 25    WISCONSIN              337    350      0151   015E
 26    WASHINGTON             351    364      015F   016C
 27    ALASKA                 365    378      016D   017A
 28    PENNSYLVANIA           379    392      017B   0188
 29    MICHIGAN               393    406      0189   0196
 30    FLORIDA                407    420      0197   01A4
 31    NORTH CAROLINA         421    434      01A5   01B2
 32    MARYLAND               435    448      01B3   01C0
 33    NEVADA                 449    462      01C1   01CE
 34    IDAHO                  463    476      01CF   01DC
 35    VERMONT                477    490      01DD   01EA
 36    OHIO                   491    504      01EB   01F8
 37    LOUSIANA               505    518      01F9   0206
 38    TEXAS                  519    532      0207   0214
 39    NEBRASKA               533    546      0215   0222
 40    ILLINOIS               547    560      0223   0230
 41    ARKANSAS               561    574      0231   023E
 42    MONTANA                575    588      023F   024C
 43    MISSOURI               589    602      024D   025A
 44    ARIZONA                603    616      025B   0268
 45    RHODE ISLAND           617    630      0269   0276
 46    KANSAS                 631    644      0277   0284
 47    CONNECTICUT            645    658      0285   0292
 48    NEW JERSEY             659    672      0293   02A0
 49    COLORADO               673    686      02A1   02AE
 50    NORTH DAKOTA           687    700      02AF   02BC
```

To help us to provide you with the best manuals possible, please make your comments and suggestions concerning this publication on the form below. Then detach, fold, tape closed and mail to us. All comments and suggestions become the property of Wang Laboratories, Inc. For a reply, be sure to include your name and address. Your cooperation is appreciated.

TITLE OF MANUAL **SORT STATEMENTS REFERENCE MANUAL**

COMMENTS:

Fold

Fold

(Please tape, Postal regulations prohibit the use of staples.)

WANG

Fold

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY CARD

FIRST CLASS PERMIT NO. 16 LOWELL, MA

POSTAGE WILL BE PAID BY ADDRESSEE

WANG LABORATORIES, INC.
ONE INDUSTRIAL AVENUE
LOWELL, MASSACHUSETTS 01851

Attention: Technical Writing Department

Fold

Cut along dotted line.

Printed in U.S.A.
13-1019

International Representatives

American Samoa
Argentina
Bahrain
Bolivia
Botswana
Brazil
Canary Islands
Chile
Columbia
Costa Rica
Cyprus
Denmark
Dominican Republic
Ecuador
Egypt
El Salvador
Finland
Ghana
Greece
Guam
Guatemala
Haiti
Honduras
Iceland
India
Indonesia
Ireland
Israel
Italy
Ivory Coast
Jamaica
Japan
Jordan
Kenya
Korea
Kuwait
Lebanon
Liberia
Malaysia
Mexico
Morocco
Nigeria
Norway
Paraguay
Peru
Phillippines
Portugal
Qatar
Saudi Arabia
Senegal
South Africa
Spain
Sri Lanka
Sudan
Syria
Thailand
Turkey
United Arab
Emirates
Uruguay
Venezuela
Yugoslavia

United States

Alabama
Birmingham
Mobile

Alaska
Anchorage

Arizona
Phoenix
Tucson

California
Culver City
Emeryville
Fountain Valley
Fresno
Inglewood
Sacramento
San Diego
San Francisco
Santa Clara
Ventura

Colorado
Englewood

Connecticut
New Haven
Stamford
Wethersfield

District of Columbia
Washington

Florida
Hialeah
Jacksonville
Orlando
Tampa

Georgia
Atlanta
Savannah

Hawaii
Honolulu

Idaho
Boise

Illinois
Chicago
Morton
Oak Brook
Park Ridge
Rock Island
Rosemont
Springfield

Indiana
Carmel
Indianapolis
South Bend

Iowa
Ankeny

Kansas
Overland Park
Wichita

Kentucky
Louisville

Louisiana
Baton Rouge
Metairie

Maryland
Rockville
Towson

Massachusetts
Boston
Burlington
N. Chelmsford
Lawrence
Littleton
Lowell
Tewksbury
Worcester

Michigan
Kalamazoo
Kentwood
Okemos
Southfield

Minnesota
Minneapolis

Missouri
Creve Coeur
St. Louis

Nebraska
Omaha

Nevada
Las Vegas

New Hampshire
Manchester

New Jersey
Bloomfield
Toms River

New Mexico
Albuquerque

New York
Albany
Fairport
Liverpool
New York City
Syosset
Tonawanda

North Carolina
Charlotte
Greensboro
Raleigh

Ohio
Akron
Cincinnati
Cleveland
Independence
Toledo
Worthington

Oklahoma
Oklahoma City
Tulsa

Oregon
Eugene
Portland

Pennsylvania
Allentown
Camp Hill
Erie
Philadelphia
Pittsburgh
State College
Wayne

Rhode Island
Providence

South Carolina
Charleston
Columbia

Tennessee
Chattanooga
Knoxville
Memphis
Nashville

Texas
Austin
Dallas
Houston
San Antonio

Utah
Salt Lake City

Vermont
Montpelier

Virginia
Newport News
Norfolk
Richmond

Washington
Richland
Seattle
Spokane

Wisconsin
Appleton
Brookfield
Green Bay
Madison
Wauwatosa

International Offices

Australia
Wang Computer Pty., Ltd.
Adelaide
Brisbane
Canberra
Milsons Point (Sydney)
South Melbourne
West Perth

Austria
Wang Gesellschaft, m.b.h.
Vienna

Belgium
Wang Europe, S.A.
Brussels
Erpe-Mere

Canada
Wang Laboratories
(Canada) Ltd.
Burlington, Ontario
Burnaby, B.C.
Calgary, Alberta
Don Mills, Ontario
Edmonton, Alberta
Montreal, Quebec
Ottawa, Ontario
Toronto, Ontario
Victoria, B.C.
Winnipeg, Manitoba

France
Wang France, S.A.R.L.
Bagnolet, (Paris)
Discheim (Strassbourg)
Ecully (Lyon)
Nantes
Toulouse Cedex

Hong Kong
Wang Pacific Ltd.
Hong Kong

Japan
Wang Computer Ltd.
Tokyo

Netherlands
Wang Nederland B.V.
IJsselstein
Gronigen

New Zealand
Wang Computer Ltd.
Auckland
Wellington

Panama
Wang de Panama
(CPEC) S.A.
Panama City

Puerto Rico
Wang Computadoras
San Juan

Singapore
Wang Computer (Pte) Ltd.
Singapore

Sweden
Wang Skandinaviska AB
Malmo
Stockholm (Solna)
Groteborg

Switzerland
Wang S.A./A.G.
Zurich
Bern
Geneva
Lausanne

Taiwan
Wang Industrial Co.
Taipei
Kaohsiong

United Kingdom
Wang (UK) Ltd.
Birmingham
London
Manchester
Richmond

West Germany
Wang Laboratories, GmbH
Frankfurt
Berlin
Dusseldorf
Essen
Freiburg
Hamburg
Hannover
Kassel
Koln
Munchen
Numberg
Saarbrucken
Stuttgart

WANG LABORATORIES, INC.
ONE INDUSTRIAL AVENUE, LOWELL, MASSACHUSETTS 01851, TEL. (617) 459-5000, TWX 710 343-6769, TELEX 94-7421

Printed in U.S.A.
700-3559G
1-81-1M