WANG

SORT STATEMENTS REFERENCE MANUAL

SYSTEM 2200

Sort Statements Reference Manual

WANG LABORATORIES, INC.
ONE INDUSTRIAL AVENUE, LOWELL, MASSACHUSETTS 01851, TEL. (617) 851-4111, 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) 851-4111. TWX 710 343-6769, TELEX 94-7421

HOW TO USE THIS MANUAL

This manual describes the use of the Sort Statements. It is assumed that the reader is familiar with the operation of the available system, has access to the following manuals: The Wang BASIC Language Reference Manual and The Programming In BASIC Manual, and/or a working knowledge of the Wang BASIC language. Additionally general knowledge of data processing techniques is desirable.

The Sort statements presented in this manual are:

MAT CONVERT	Reformats data from the normal storage format into the hexdecimal sort format and stores them 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. The data can be copied in forward or reverse directions.
MAT MERGE	Creates the locator-array (merge array) used to merge alphanumeric array rows into a single (sorted) array (or file).
MAT MOVE	Moves data from one alphanumeric array to specified locations in another alphanumeric array in the sorted order specified by the locator-array which contains the subscripts of elements in the desired order. Elements can be moved element-by-element, or by fields within each element, to the desired output device.
MAT SEARCH	Searches an alphanumeric array for character strings that satisfy a given relation; element boundaries are ignored. The search can be performed at specified intervals and a variable number of characters can be compared.
MAT SORT	Creates the locator-array pointers needed for sorting the elements of an alphanumeric array into ascending or descending order; leaves the subscripts of the ordered data in the output locator-array.

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 utilities rather than doing their own programming with the Sort Statements. Users who have a Wang disk unit and wish to do sorting should refer to the Disk Sort Utility Reference Manual. Under certain conditions a user may not need to prepare sorted files, but merely to produce lists of sorted data. Another Wang-supported utility, KFAM (acronym for Keyed File Access Method), provides this capability. With the random access method of KFAM, the user can set up a file on which Sort Keys and pointers to the disk addresses of his data file are both stored. (See the KFAM Reference Manuals). Finally, ISS (INTEGRATED Support System) KFAM & SORT Utility programs provide extended range in multiplexed disk environments.

TABLE OF CONTENTS

CHAPTER 1: GENERAL INFORMATION

1.1 INTRODUCTION

The Sort Statements are a group of programmable statements that provide high speed data sorting, searching, and moving capabilities.

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 sort time and program size (MAT CONVERT, MAT SORT, MAT MOVE, MAT MERGE), and those of general use to move or search character or data strings (MAT COPY and MAT SEARCH). The last two can be useful in text editing, statistical and random access data retrieval applications and treat alphanumeric arrays as groups of characters without regard to array element boundaries.

In a manual of this type it is impossible to enter into a discussion of all the sorting and merging techniques that have been invented. The examples have been chosen to provide alternatives and illustrate internal and external sorts. An internal sort is one in which all sorting takes place inside the memory of the computer; an external sort is performed when memory is insufficient to take the entire file to be sorted at one time. In such a case, various peripheral devices can be used for storage of permanent and intermediate data files. In practice, the most common sort or sort/merge techniques are of this latter type.

Examples have been provided to illustrate the rules and the use of the Sort statements. Remember when using these statements that subscript locate mode arrays are automatically created by MAT MERGE, MAT SEARCH and MAT SORT. Locate mode is a means to access data by pointing to its location instead of moving it. To do a sort, for example, it is possible and very fast to read in an array, execute MAT SORT to get the ordered pointers for all elements in the array, and then execute another Sort statement (MAT MOVE) to place all the elements in the correct order on an output device. Compared to the usual sort-and-exchange or bubble sort techniques, the ease of use and rapidity of locate mode excels. This feature opens up important applications where rapid access to large amounts of data is desirable. For example, in statistical analysis it may be necessary to pass several times over the same mass of data to cull out information according to several parameters. The Sort statements can do this swiftly without the necessity of physically dumping out all the data each time.

1.2 INSTALLATION

The Sort Statements are a standard feature of most 2200, WCS, PCS systems. They are available as Option 5 for the 2200B and 2200C processors (Option 5, Sort Statements, cannot be installed in a processor that also contains Option 1, Matrix Statements) and as Option 24 for the 2200S and WCS/10 processors. For the 2200VP, see the sort statements in the BASIC 2 Language Manual.

For processors that do not include Sort Statements as a standard feature, the Sort Statement option can be installed at the factory or retrofit by a Wang Service Representative.

1.3 ARRAYS

The input files for the Sort Statements must be explicitly blocked (i.e., written in either alphanumeric or numeric array form).

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

An array is a number of mathematical elements arranged in a specified order. Arrays can be either one- or two-dimensional. A one-dimensional array can be thought of either as a single row or a single column. A two-dimensional array is arranged in rows and columns. In this arrangement each element can be uniquely identified by its position. The usual array notation describes position with subscripts. A one-dimensional array has a single subscript and can also be called a vector. A two-dimensional array has two subscripts which are always for "row, column", never "column, row". A two-dimensional array is like a table (see Figure 1-1).

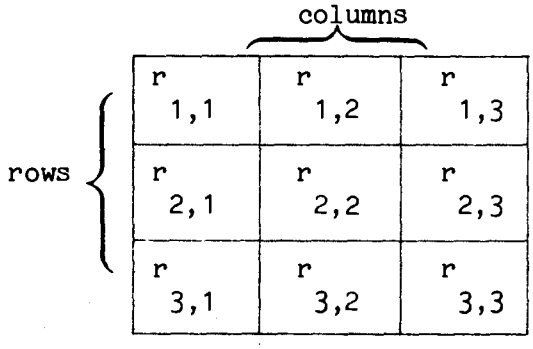


Figure 1-1. A Two-Dimensional Array

For example, in the following array the value in element (1,2) is A and the value in element (3,4) is B.

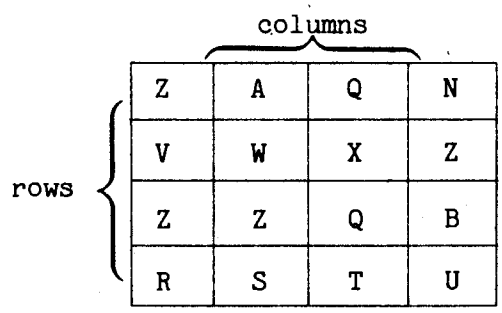


Figure 1-2. An Alphanumeric Array

Arrays can contain either numeric or alphanumeric elements as defined by the assigned array name, e.g., either A() or A$(), respectively. Every array must not only be appropriately named, but must also be properly dimensioned with a DIM or COM statement. For example, to define and dimension the array illustrated in Figure 1-2, the following statement suffices:

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

For one-dimensional numeric arrays, the available memory size is the only size restriction. For two-dimensional arrays, a second restriction is imposed in addition to memory size. The system uses a single byte to represent each subscript of a two-dimensional 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. This restriction can be expressed in a different way by stating that each row in a two-dimensional array is limited to a maximum of 255 elements or each column is limited to a maximum of 255 rows. The maximum total number of elements in a two-dimensional array is 4,096 elements (bytes). This is equivalent to 32,768 bits.

For alphanumeric-arrays, the length of each element is not fixed by the system. Instead, it can be set by the programmer to any length between one byte and 124 bytes, inclusive; for example:

```
10 DIM A$ (4,4) 124
```

would specify a 16 element array with 124 bytes in each element (1984 bytes total). In this case, as with numeric arrays, the determining factor in restricting array size is the available memory space.

In order to indentify each variable in memory, the system automatically inserts several bytes of control information at the beginning of the variable area. These additional bytes are completely "transparent" to the programmer (that is, they are used exclusively by the operating system, and cannot be accessed by the application program), and they represent a fixed overhead for each variable defined in a program. The number of control bytes required differs according to whether the array is alpha or numeric.

Note:

Subscripts defined in DIM statements must be positive integers (1 to 255). Zero is not allowed as a defining subscript. Once defined, variables can be used as subscripts. Space required for an array must not exceed 4096 bytes. In some Sort statements, a subset of an element can be defined. Numeric array elements are set to zero, and alphanumeric array elements to spaces (HEX(20)), when memory space is initially allocated, dimensioned. Mat Sort executes fastest when the array length is one greater than a power of two. Block length = row length in a merge array. The optimum number of elements per block (N) in a sort array will be any logical multiple of (2^X+1). The worst case occurs when the number of elements (N) is an exact power of two. See Appendix B.

1.4 INTERNAL STORAGE

All data is physically stored by the Wang System in eight byte fields. The format in which data are stored internally in your system depends upon whether the data to be stored are numeric, alphanumeric, or MAT converted. Alphanumeric data are stored as a series of bytes where each byte contains a single character. A numeric value is stored in exponential form. The first half of the first byte contains a code for the signs of the mantissa and the exponent as follows:

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

The next two half-bytes contain the exponents HEX digits ($-99E \leq E \leq +99$) and the remaining 6 1/2 bytes contain the 13 digits of the mantissa (see Figure 1.3), one digit in each half-byte.

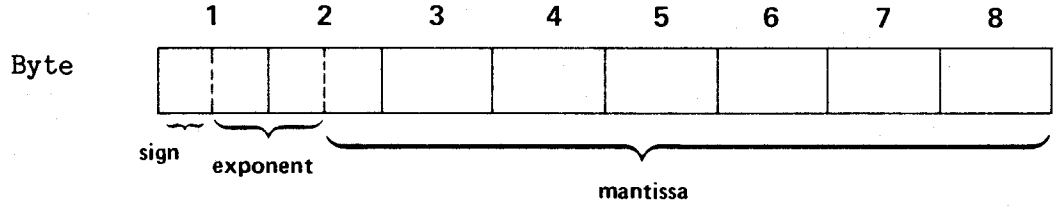


Figure 1-3. The Eight Byte Storage Field

When operated on or output by Sort statements, the work arrays will consist solely of hex codes. Only arrays in the mat convert mode can be operated on by the sort set. This is because sorting requires a special relationship between a numbers sign and exponent to describe its magnitude. For example:

		0	
-1.0	-0.1	+0.1	+1.0
mantissa (-) exponent (+)	mantissa (-) exponent (-)	mantissa (+) exponent (-)	mantissa (+) exponent (+)

In each instance, one (1) is represented by the same hexcode (31). Therefore the sign and exponent codes must precede the code for one (1) in order for the computer to recognize the differences in magnitude and properly sort these four values. Note the difference in the storage format's of alphanumeric, numeric and MAT converted data. Because of this, Appendix D containing alphanumeric to hex code conversions is provided.

Numeric values are stored sequentially ascending in sort format. If the elements of the alpha array contain less than eight bytes, the stored value is padded with spaces (at the right). If the elements of the alpha-array contain more than eight bytes, the value is truncated; least significant digits are lost. The truncation feature can be useful for conserving memory when it is known that all values to be MAT CONVERTed have few significant digits.

In the MAT converted format the first four bits (half-byte) of each value in the alpha-array are used to represent the signs of the mantissa and the exponents as follows:

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

The next eight bits (two half-bytes) are used to represent the high and low order digits of the exponent. They are given in decimal or nines complement form to indicate the size of the value as follows:

form	indicates
decimal	mantissa and exponent both positive
complemented	mantissa positive, exponent negative
decimal	mantissa and exponent both negative
complemented	mantissa negative, exponent positive

(i.e., exponents are given as their nines complements if the signs of mantissa and exponent differ). The remaining bytes of the value are the digits of the mantissa. These digits are 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.

Values in sort format can be stored in a specified field of each alpha-array element by using the limiting field expressions (s,n). The number of bytes specified in each field must be at least 2; if more than eight bytes are specified, the value is truncated at the right, 2 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.

1.5 SORT/MERGE OPERATIONS

There are many possible sorting techniques which can be used to order data as needed. A general technique that utilizes the MAT CONVERT, MAT COPY, MAT MOVE, MAT SORT and MAT MERGE statements is described below. It contains a Sort Phase and a Merge Phase,in that order, since merging can only be done on sorted data. The user who also wishes to explore those methods using Random Access Data Files (on disk) should refer to the KFAM Reference Manual.

The Sort Phase

In this phase, data records to be sorted are read from a data file (on tape or disk), perhaps transferred for efficiency in memory using MAT COPY, and ordered in strings (small sorted groups) on the basis of a Sort Key. The Sort Key is a field (or fields) from the data record which is used to do the ordering; a string is a sub-file of records which are stored in sorted order. For example, to sort a file by customer number and invoice date, the Sort Key can contain any number of variables or fields from the data record, even the entire record if necessary. Using a short Sort Key improves running time because comparisons between the keys for sorting are done many times in the process of a sort.

In building the Sort Key, it is essential that both numeric and alphanumeric data be compared in the same operation, character-by-character. All numeric values are stored in a floating point exponential format optimum for numerical calculations; these values must be converted for sorting purposes to an appropriate alphanumeric equivalent. MAT CONVERT does this job. 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.

In the Sort Phase, the number of records and/or keys initially put into each string is usually determined by the amount of memory available, although a string is not restricted by memory capacity. Subsequently, the records are read in groups, Sort Keys are built for each, and the records and/or keys are sorted into strings and stored (on disk or tape) in at least two Merge Files. MAT SORT is used to do the job of sorting the initial data groups into (ordered) strings. To do this, MAT SORT sets up a subscript array called a locator-array whose elements point in sorted order to the locations of the records which are in memory. MAT MOVE must then be used to interrogate the locator-array and actually move the data into sorted order on an output medium (tape or disk) in files to be merged (Merge Files).

The Merge Phase

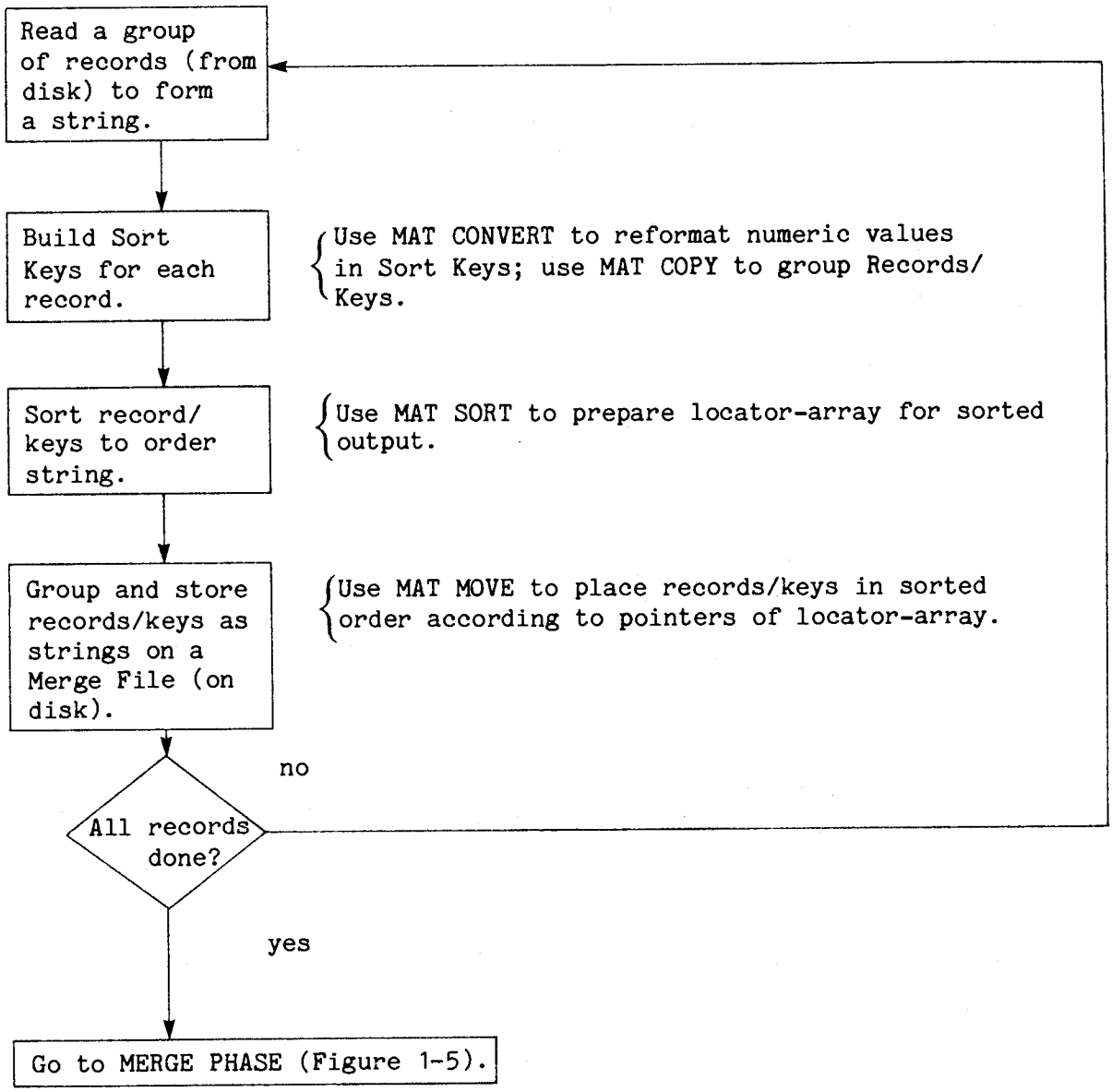
The Merge Phase generally consists of a number of merge passes. In each pass, strings from two or more Merge Files are read, combined into (longer) strings and stored on two or more new Merge Files. The process is repeated until a single string containing the entire file in sorted order is created.

> Note:
>
> It is not necessary for a given string to be completely stored in memory at one time during the merge process; it can be loaded in segments.

MAT MERGE is normally executed a number of times in each merge pass. For its use, a two-dimensional alpha array is set up to receive the string segments being merged. Each row of the array represents one string segment from one input Merge File. The strings contain the records or keys on which comparisons are made.

When MAT MERGE executes for the first time, 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 pre-defined locator-array and this key is eliminated from future comparisons. MAT MERGE is then executed again seeking the lowest key not yet merged. MAT MERGE execution terminates either when a segment from a Merge File (a row) is exhausted or when the locator-array is full. MAT MOVE must then be used to move the records being merged into an array where segments of the output file are being formed. Once an input segment has been exhausted, its corresponding row can be refilled with MAT COPY. Once an output segment is full it can be written out onto a new Merge File for later use. This entire process is repeated until all strings from the old Merge Files have been exhausted and a new merge pass can begin. Merging continues until a single sorted string containing all records from the original input file has been created.

A block diagram for a typical Sort program using Sort statements can be found in Figure 1-4, and for the Merge Phase of a sort routine, in Figure 1-5.



Note:

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

Figure 1-4. Block Diagram of a Typical Sort Program, Sort Phase

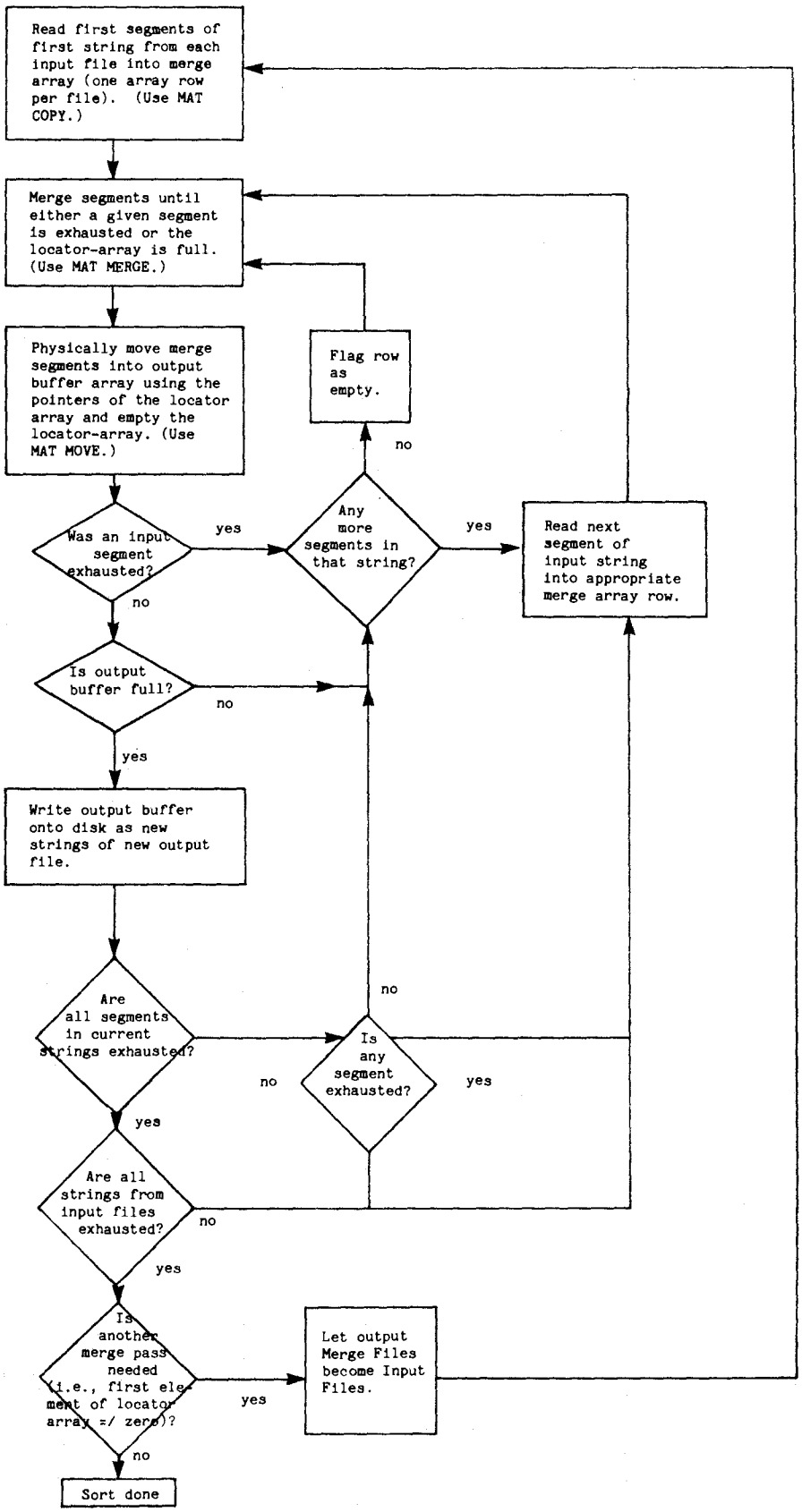


Figure 1.5 Block Diagram of the Merge Phase of a Sort Program

1.6 DESCENDING SORTS

Sometimes it is necessary to sort some of the variables in a Sort Key in descending order. This can be accomplished by inverting the order of an ascending sort or by complementing (using the XOR statement) those characters in the final Sort Key which represent descending order key variables. For numeric data, complementing must be done after data have been put in sort format using MAT CONVERT.

Example 1 A$(), is a Sort Key to be complemented.

```
100 XOR (A$(),FF) does the job.
```

Example (B$(1), is an array element representing a Sort Key and record; its sixth through tenth characters are part of the Sort Key to be complemented for a descending sort.

```
200 XOR (STR(B$(1),6,5),FF) does the job.
```

EXAMPLES

Example 1. Using MAT SORT and MAT MOVE

In the following short program an input or sort array I$() is processed by MAT SORT and the locator-array L$() containing pointers is filled. 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
```

Try it. 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
? S
? D
? E
? A
? Z
? X
? Y
? G
? B
? J
0401090102010301080l0A0101010601070l050100002020
ABCDEGJSXYZ
```

Figure 1-6. Output from Example 1

Example 2. An External Sort

In this example there are three sorted data files on disk (INPUT1, INPUT2 and INPUT3). Each has logical records which contain 50 elements; each element contains 8 bytes. The program illustrated 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 hence to the output file.

The scheme for merging proceeds as follows:

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

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 is used to empty the locator-array and to move data from the merge array to the output buffer. The movement of data ceases when the locator-array has been emptied or 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 must be 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 buffer is checked (with the VAL function) to see if a row in the merge array has been completely scanned. If all its elements have been used, it is refilled with data from the input file and the appropriate element in the primary work array is reset to 1.

8. If there are no more records to be input, normal operation continues.

9. 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.

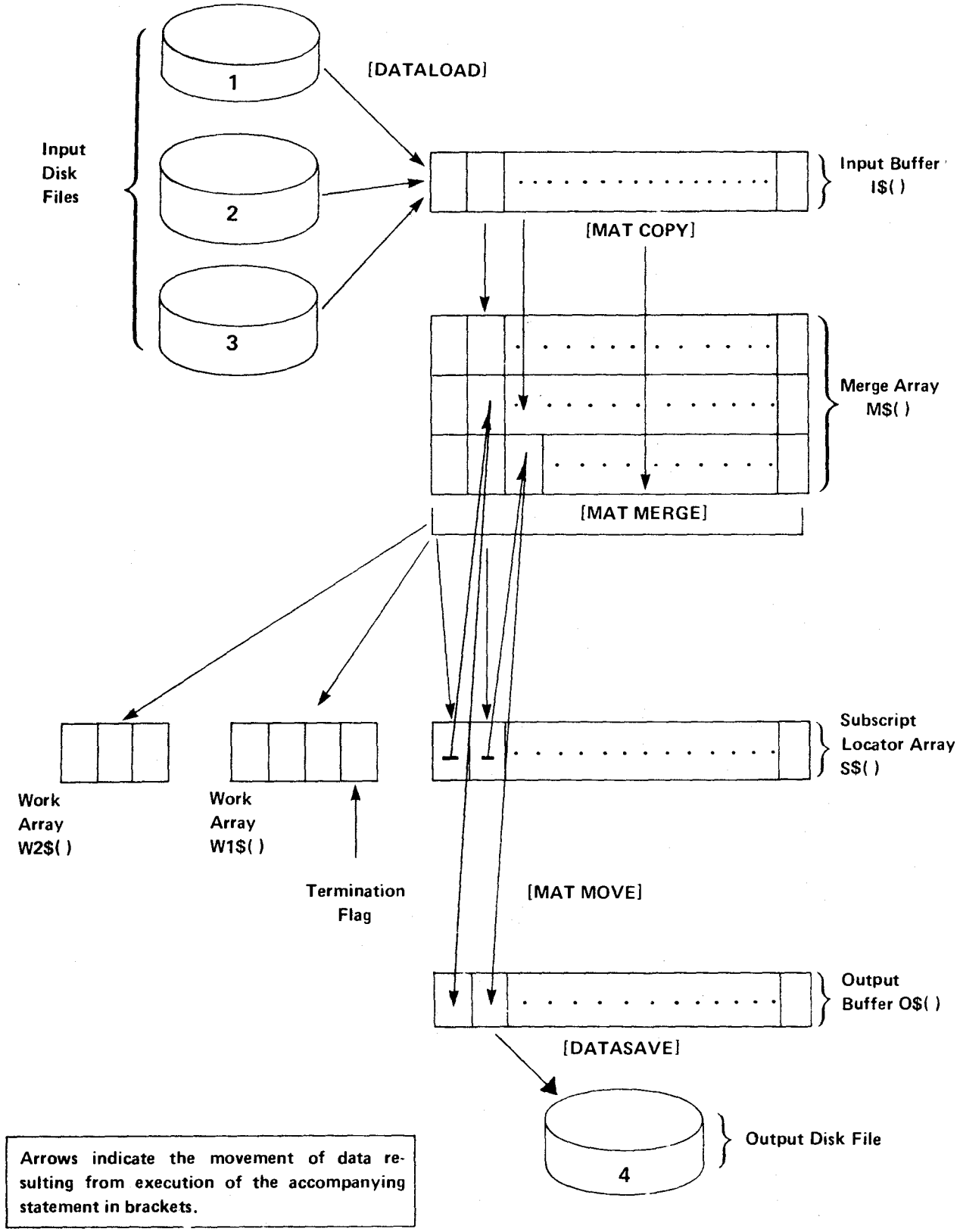


Figure 1-7. Schematic Diagram of MAT MERGE External Sort

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 3. An Internal Sort

In this example, two arrays I$#1 and #2 are merged. Their data are given in a DATA statement. Only a listing of this routine is provided, followed by a printout of the data which are displayed on the CRT. It is recommended that this routine be entered on your system and run, since the CRT display tells much of the story of MAT MERGE. The routine uses MAT COPY to fill the input array A$(), 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. Checks are made to determine whether to terminate merging, and the merge operation continues. When the first element of the locator-array is still zeroes 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; for those that cannot the HEXPRINT statement must be used. This routine requires only a CPU with Sort statements, a keyboard and a CRT.

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) 5,0$ (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=1TO 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
HEXPRINT W1$ (I) :NEXT I:STOP
360 REM   MOVE THE MERGE ARRAY TO THE OUPUT BUFFER 0$ 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 0$ (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. 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) =Q$ (I) :NEXT I
460 MAT COPY 0$ () TO K$ () <(R-1)*20+1,20>
470 P=P+N:Q=Q+N:S=N+1:PRINT "P=",P,"Q=",Q,"S=",S
```

Program Listing (Continued)

```
480 IF S<5 THEN 380:REM   RETURN TO MOVE IF MORE DATA TO MOVE
490 REM         DISPLAY THE CONTENTS OF 0$
500 PRINT "OUTPUT ARRAY BUFFER"
510 FOR I=1TO4:PRINT J$ (I) ,:HEXPRINT 0$ (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) =0$ (I) :NEXT I
590 FOR I=1 TO 8:PRINT K$ (I) :NEXT I
600 FOR I=1TO 4:PRINT J$ (I) ,:HEXPRINT 0$ (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

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
```

Program Printout (Continued)

```
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$=
0103
0104
0000
0000
WORK VECTOR FROM MAT MERGE IS NOW:
FF
03
01
COUNTERS AND THE OUTPUT BUFFER ARE:
M=                1                 S=
N=                4
N=                2                 ELTS. WERE MOVED
T=                1
MAT MERGE HAS BEEN EXECUTED
THE LOCATOR ARRAY L$=
0203
0204
0000
0000
WORK VECTOR FROM MAT MERGE IS NOW:
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
```

Program Printout (Continued)

```
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 provides a discussion of syntax and examples using the Sort statements. As with the 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 set down in Chapter 2 of this manual are not followed exactly, programs using 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 represent actual characters in the statement; lowercase words represent parameters that can change.

No array to be used with the Sort statements can contain more than 4096 bytes. Note that dimensional rules must be followed; the locator-array must always have elements of length 2.

> Note:
>
> It is essential that any data set containing numeric values be passed through a MAT CONVERT operation before being used with the MAT SORT or MAT MERGE statement.

> Note:
>
> 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 on an array that is subsequently operated on by a Sort Statement only if the total number of bytes in all elements after the MAT REDIM is the same as before the MAT REDIM.

MAT CONVERT

General Form:

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

where (s,n) are expressions designating a field within each element of the alpha-array used to store the converted value.

s specifies the starting position of the field.

n specifies the number of bytes in the field.

Note:

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 so that the resulting alpha-array-elements can be sorted (by MAT SORT or MAT MERGE) in proper numeric sequence. The format of data passed through the MAT CONVERT statement is called sort format.

Examples of Valid Syntax:

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

Working Example:

In the routine below, the numeric-array N() contains four numeric elements; they are passed to A$() and to B$() according to the 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) =12345678: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
```

Figure 2.2 A Program Using MAT CONVERT

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

```
NUMBERIC ARRAY,N():
                  123
                 -456
                   12345678
                  0
ALPHA ARRAY, A$():
                902123                    097543
                898123                    800000
ALPHA ARRAY, B$():
                2090212320202020 ⎫
                2009754320202020 ⎬ HEXADECIMAL
                2089812320202020 ⎪
                2080000020202020 ⎭
                ‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾
                   FIELD(2,3)=elements 2, 3 and 4
```

Figure 2-3. Output from Program Using MAT CONVERT

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

```
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$,2)
1120 AND (Z3$,F0)    : REM 2ND EXP. DIGIT
1130 AND (Z2$,0F)    : REM FIRST EXPT. DIGIT
1140 OR (Z3$,Z2$)    : REM EXP. REVERSED
1150 ROTATE (Z3$,4) : REM EXPONENT
1160 REM     NOW DETERMINE THE SIGN
1170 AND (Z1$,F0)  : REM SIGN BITS
1180 IF Z1$ = 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$,1,1),0F)
1330 REM INSERT SIGN
1340 OR (Z$,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
```

Figure 2-4. The "Unconvert" Routine

MAT COPY

General Form:

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

where s,n are expressions defining a portion of either array. s specifies the position of the starting byte to be used; n specifies the number of bytes to be used.

s must be $0 < s \leq$ bytes-in-the-array.
n must be $0 < n \leq$ (bytes-in-the-array - s+1).

Each array is treated as a contiguous string of bytes, ignoring element boundaries. If s,n are omitted, the entire array is used.

Purpose:

MAT COPY transfers data from the source alpha array to the output alpha array byte-by-byte. The source and output arrays can be the same 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, combine or divide elements, move data within arrays, and using the (-) inverse parameter store data in reverse order within arrays.

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. A portion of an array can be specified by using the limiting expressions (s,n). "s" is the counted position of the first byte to be used in the array. The position is found by counting bytes across the first row (left-to-right), then across the second row, etc., until the desired byte is encountered. "n" specifies the number of bytes to be used at one time.

If a minus sign (-) precedes the input array name (or its specified portion), data are taken from it in reverse order and stored left justified. In this case, 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>
```

Examples: DIM A$(5)1, B$(7)1

A$() =

A	B	C	D	E

MAT COPY A$() TO B$()

B$() =

A	B	C	D	E	space	space

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

B$() =

space	space	E	D	C	B	A

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

B$() =

E	D	C	B	A	space	space

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

B$() =

space	space	A	B	C	D	E

Working Example:

In this example, A$() has been dimensioned as a (5)1 element array, and B$ as (7)1. If A$() contains the characters A to E, and these data are copied to B$(), the result is as shown in Figure 2-5.

```
10 DIM A$(5)1, B$(7)1
20 FOR I=1 TO 5 : READ A$(I) : NEXT I
30 MAT COPY A$() TO B$()
40 DATA "A", "B", "C", "D", "E"
50 FOR I = 1 TO 5 : PRINT A$(I) : NEXT I
60 PRINT ""
70 HEXPRINT B$()

ABCDE
41424344452020
```

Figure 2-5. Output from MAT COPY

General Form:

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

Note:

1. The merge-array must be a two-dimensional alpha-array with not more than 254 rows or columns; its number of rows (n) must be≥1.

2. Work-vector-1 must be one-dimensional. It must have at least as many elements as the merge-array has rows (+1); each element must have length = 1. Each of its elements must be initialized to HEX(01) before initial use of the MAT MERGE statement.

3. Work-vector-2 must be one-dimensional; it must have at least as many elements as the merge-array has rows each element must have length = 2. It does not need to be accessed by the user but must be defined.

4. The larger the locator-array, the faster the merge. The locator-array must be one-dimensional; it should have at least as many elements as there are in a single row of the merge-array. Let it have as many elements as possible. Each of its elements must have length = 2.

5. Each row of the merge-array must be pre-sorted.

Purpose:

MAT MERGE enables the programmer to quickly combine several ordered data file into a single large ordered file. Each row of the merge-array represents a set of sorted data elements (one datum per element) to be merged. Each row is essentially a merge buffer for a given input file. A merge operation is done in a series of passes, each pass being the execution of a MAT MERGE statement. 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 merge operation halts when either:

1. the locator-array is filled with subscripts

   HEX 00

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

   HEX 01 ---- HEX FF (row 0 to FF)

At this point the program processes the data merged so far, and if necessary, replenishes an empty row (in the merge array) from disk or tape. MAT COPY can be used to refill a row, and MAT MOVE to move merged data to an output buffer array.

The remaining elements in work-vector-1 are used to indicate the current status (pointer to next element to be used) in each row of the merge array. Before initial execution of MAT MERGE, the merge array must be filled with data (using MAT COPY) and all elements of work-vector-1 must be initialized to hex(01) (using INIT).

Each ith element of work-vector-1 receives the subscript of the next element in the ith row to be compared with elements in the other rows. When a row has been completely scanned, hex(FF) is placed in its corresponding work-vector element. The program must then refill that row with data from the corresponding data file and reinitialize the respective subscripts in work-vector-1 to hex(01). 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 input array, hex(FF) in the corresponding element of the work-array indicates that MAT MERGE is to ignore that row. When MAT MERGE determines that all rows are to be ignored, hex(0000) is returned to the first element of the locator-array and execution of the current merge pass is complete.

Whenever MAT MERGE begins execution, the locator-array is filled with the subscripts of the merged data beginning with the first element. The locator-array must therefore always be emptied before execution of MAT MERGE. Once MAT MERGE has executed, the locator-array should be used to transfer the specified data to another output array or file. If the locator-array has not been filled by the execution of MAT MERGE, the element following the last valid subscript contains hex(0000).

Examples of Valid Syntax:

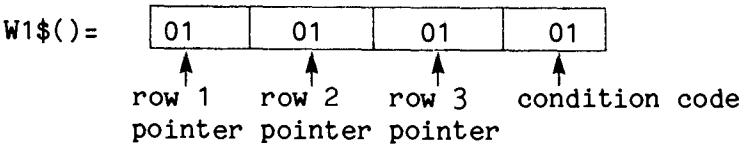
```
10 DIM A$(10,50)10,W$(11)1,W1$(10)2,S$(50)2
20 MAT MERGE A$() TO W$(), W1$(), S$()
```

Example 1. A Three-row Merge

In the following example, the three rows of the merge array A$ are merged using MAT MERGE. Each of the rows may be thought of as a merge buffer. Prior to the merge, each row must be sorted; in the case illustrated, 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 W1$ contains pointers, the starting/current element in each row to be scanned for the merge; each of its elements is initially set (using INIT) to hex(01). Each of its first three elements corresponds to a row of the input array, as follows:



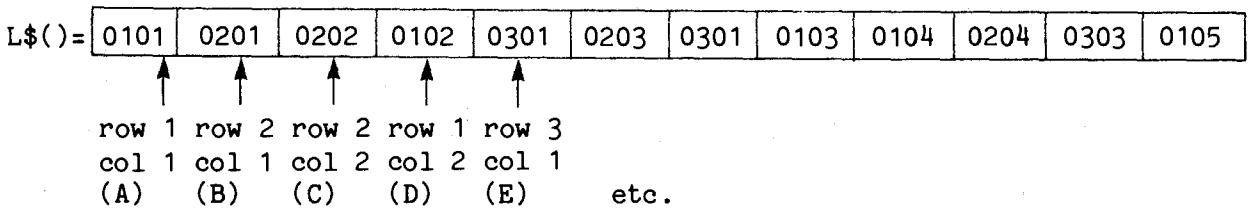
The last element receives a code from MAT MERGE to indicate how the merge pass halted; its value prior to execution of MAT MERGE is unimportant.

The following statements, given the arrays A$() and W1$() as described above:

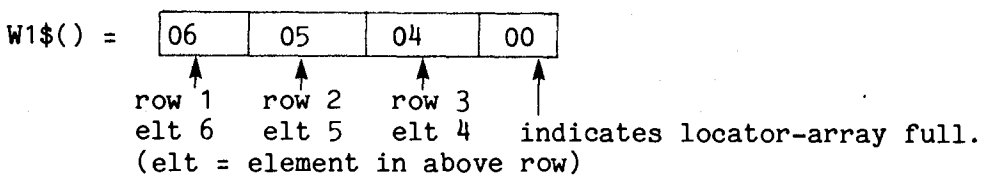
```
10 DIM A$(3,6)1, W1$(4)1, W2$(3)2, L$(12)2

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

produce the following merge results.

The subscript locator-array L$ is filled with the subscripts of the twelve lowest-order elements from the three rows; the subscripts are placed in the locator-array following the order of the merge-array as follows:



Work-vector-1 W1$ is rewritten to indicate which element in each row is to be taken in the next merge pass. Since MAT MERGE halted because the locator-array was full, the last element of W1$ is set to zero (00). W1$ thus contains:



The twelve records indicated in the locator-array L$ can now be moved to an output array or file and row 1 of the merge-array can be refilled for reexecution of the MAT MERGE statement. If row 1 is refilled with new data, the current-element position for row 1 in work-vector-1, W1$(1), should be reset to hex(01). If there are no more data to fill the row, the current-element position for the row should be left as hex(FF). When MAT MERGE is again executed, the row will be ignored.

Example 2. Termination of MAT MERGE with Emptying of Merge-Array

This example illustrates termination of MAT MERGE execution when a row of the merge array has been exhausted. Given a 3 by 6 merge array A$():

A$() =

row/col	1	2	3	4	5	6
1	A	B	E	F	H	I
2	C	G	J	K	N	Q
3	D	L	M	O	P	R

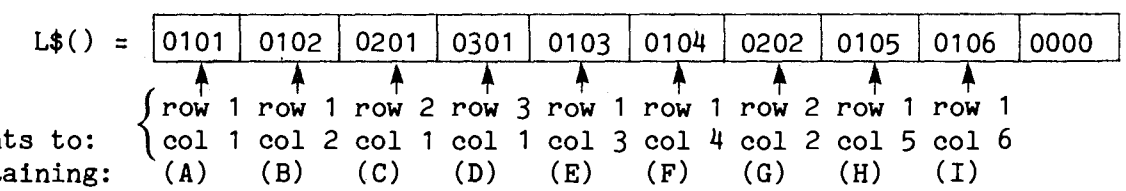
All elements in the four-element work-vector-1 are initialized to hex(01) using INIT, to indicate the starting column of each row as 1. W1$, work- vector-1, thus contains:

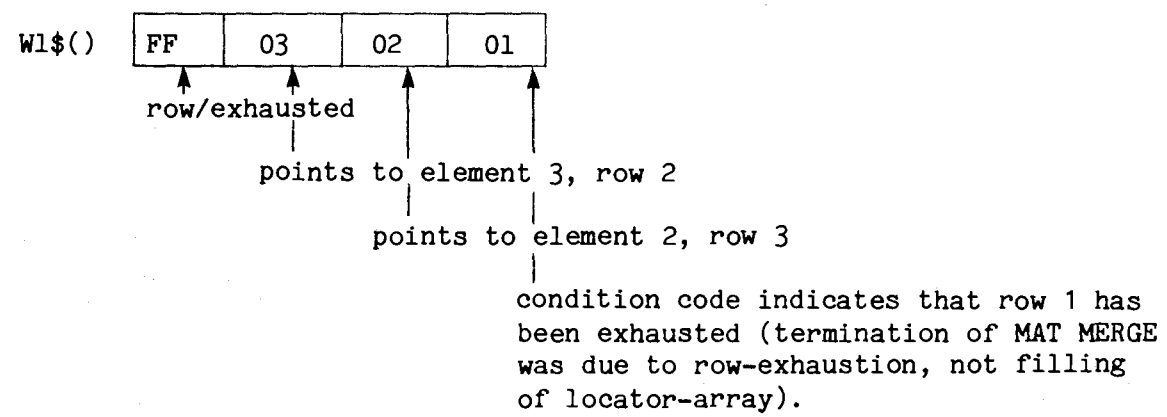
element	contents	function
1	01	row 1 pointer
2	01	row 2 pointer
3	01	row 3 pointer
4	01	condition code

The Dimensioning and MAT MERGE statements:

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

when executed once create two arrays as follows:





In L$(), the locator-array, hex(0000) in the tenth element indicates the first unused position of the locator-array. The locator-array is filled with the subscripts of the nine lowest elements in the merge array. Elements from all three rows in the merge array are used because MAT MERGE continues to extract subscripts until all the elements in row 1 have been exhausted, since the locator-array contains more elements than any row of the merge array. Pointers to subsequent elements in each row are stored in work-vector-1; since row 1 was exhausted (all elements read), the pointer for row 1 is set to hex(FF). At this point the data pointed to by the locator-array must be moved to an output array (using MAT MOVE). Then either row 1 of the merge array can be refilled and the row 1 pointer in W1$ reset to hex(01), or merge passes can continue leaving W1$(1)=FF if there are no more data to add. With the element in W1$() left as FF, the row is ignored in subsequent processing.

MAT MOVE

General Form:

MAT MOVE has two general forms:

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

and

MAT MOVE source-numeric-array-designator, starting locator-array-element [,m] TO output-numeric-array-element

(s,n) are expressions defining a field within each element of the source-alpha-array; s specifies the starting position of the field within an array element, while n specifies the number of bytes in the field.

m is a numeric scalar variable specifying the maximum number of bits to be moved.

m must be $0 \leq m \leq 32767$. When MAT MOVE executes, the number of elements actually moved is returned to this variable. The starting locator-array-element is the first element to be used from the locator-array. The locator-array is normally constructed by a MAT MERGE or MAT SORT statement.

Purpose:

Used to transfer data from one array into another MAT MOVE effectively orders data by moving it element-by-element from one array to another in the order specified by the subscript locator-array. Source and receiving arrays can be either alphanumeric or numeric so long as they are not of different types.

Data is directly moved from the move-array beginning with the first element to the receiver-array beginning at the specified receiver-array-element if a locator-array is not specified. If the locator-array is specified, data is moved indirectly from the move-array in the order given by the subscripts in the locator-array, starting with the subscript in the specified locator-array-element, or in the first element of the locator-array, if no element is specified.

Data is transferred into sequential bytes in the receiver-array, beginning with the specified element, row by row. If both the move-array and receiver-array are alphanumeric, data may be moved to and from 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 is an array of row/column subscripts which point to the data in the source array that are to be moved to the output array. The first subscript used is stored in the element specified as the starting locator-array element. The first element in the output array to receive data is specified as the 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. If no locator-array is specified, data is transferred sequentially from the move-array, starting with the first element, into sequential elements of the receiver-array, beginning with the specified receiver-array element, row by row. Either the move-array or the receiver-array or both are alphanumeric, data may be moved to and from designated fields of each element. 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 values in the source-array are shorter (in number of bytes) than the values in the output array, values are padded with spaces at the right; if values are longer, 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,AI$(100)5,A2$(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)
```

Examples of working programs:

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

A$() =

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

This is the source array to be moved.

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)

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

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.

GENERAL FORM:

MAT SEARCH search-alpha-array-desig. [<s,n>], { < | <= | = | > | >= | <> } alpha-variable TO location-array-desig. [STEP expression]

where<s,n>are expressions defining a portion of the search-array. s specifies the starting position of the first byte to search; n is the number of bytes within the area to be searched. "s" is determined by counting bytes left-to-right across the first row, left-to-right across the next row, and so on, ignoring element boundaries.

$0 < s \leq$ bytes in the search-array.

$0 \leq n \leq$ (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.

Purpose:

MAT SEARCH scans the search-array for substrings that satisfy the relation defined and stores the location of each such substring in the location-array. The locations are stored in the order in which they are found and are stored as two-byte binary values 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 numeric expressions (s,n). If these expressions are omitted, the entire array is scanned. The search-array is treated as a single contiguous character string, i.e., element boundaries are ignored. The length of the substrings in the search array equals the length of the value of the alpha-variable compared.

Note:

Trailing spaces are not considered to be part of any alpha-variable-value being compared. If it is necessary to check for trailing spaces (hex(20)), 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 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 to be used is filled with zeroes (hex(0000)).

If the STEP parameter is not used, MAT SEARCH starts at the first character in the search-array and checks to see if the character string starting there satisfies the given relation. If so, the location of the string is stored in the first element of the location-array, the substring starting at the second character is checked, and so on. If, however, the STEP parameter is used, the position of the next substring to check is the position of the last substring checked + 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.

> 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.

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: Given array G$() as follows:

row/col =

G$() =	1	2	3	4
1	B	A	C	D
2	C	B	E	A
3	A	F	A	E

and a location-array P$() dimensioned P$(2,6)2, when the statements:

```
20  INIT (FF) P$() : V$="A"

30  MAT SEARCH G$(), =V$ TO P$()
```

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

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

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

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

where:

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

MAT SORT

General Form:

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

Note:

The work-array and the locator-array must have at least as many elements as the sort-array; their elements 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 subscripts arranged according to the ascending order of elements from the sort-array. Subscripts in the locator-array are two-byte values; the first byte is the row subscript, the second 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)).

The locator-array can be used (with MAT MOVE) to create a new data array in sorted order.

Note:

Sort-array elements of identical value have contiguous elements 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$()
```

The following routine takes a sort-array of twelve elements (G$()) and creates a subscript array G1$().

If G$() =

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

the statements:

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

create the following locator (subscript) array when executed:

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.

The example on p. 9 also illustrates the use of MAT SORT.

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 any other usual character, 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 which will ensure that the unused elements of any array will sink to the bottom (end) of the sorted array.

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.

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. Worst case results occur when n is a power of two (i.e., $n=2^X$); best case, when $n=(2^X+1)$ 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 together.

When using both MAT SORT and MAT MERGE, while each additional row of data added to the merge array causes a linear increase in the time required for sorting (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 'sweetspots' 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

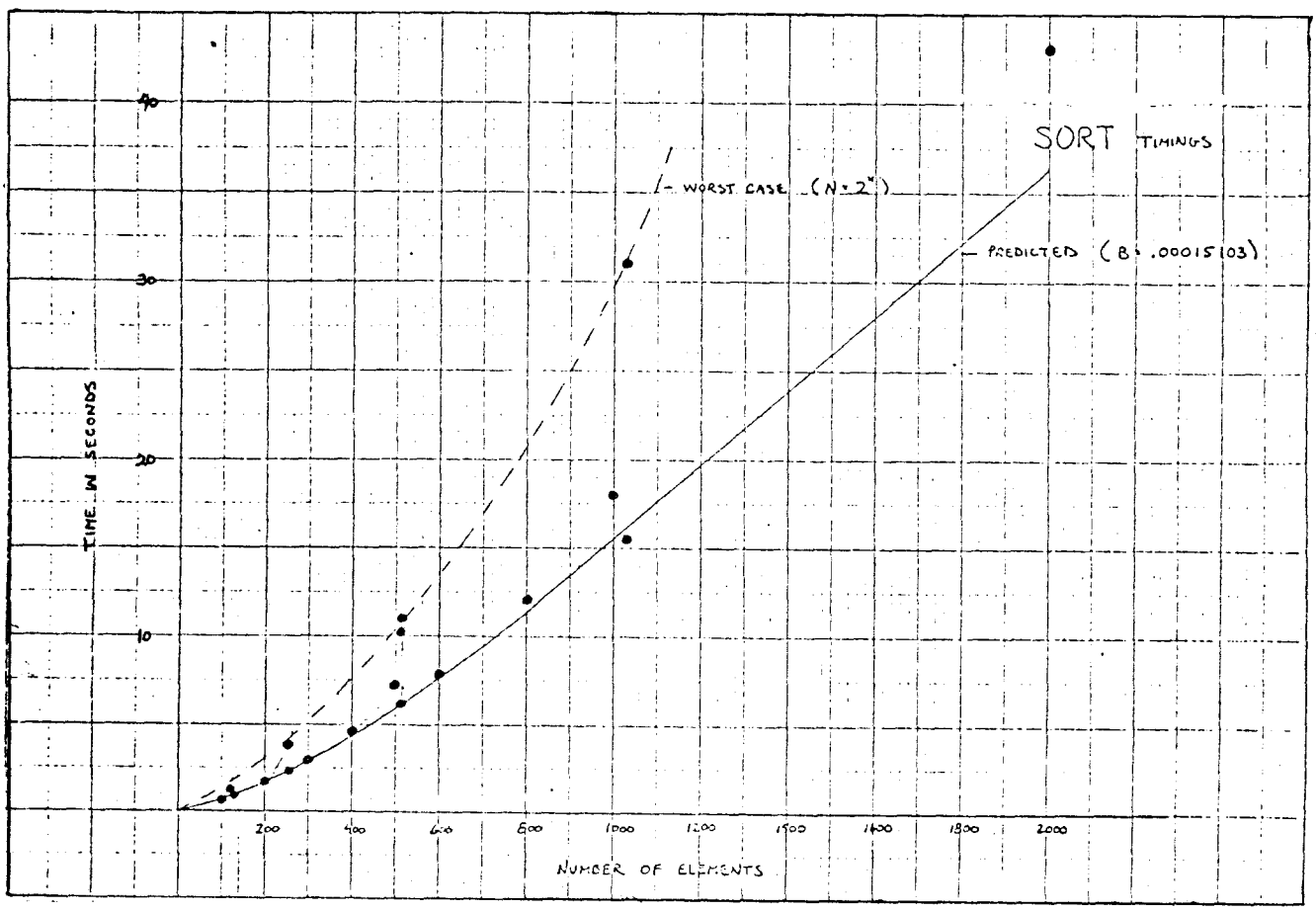


Figure B-1. MAT SORT Timing

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

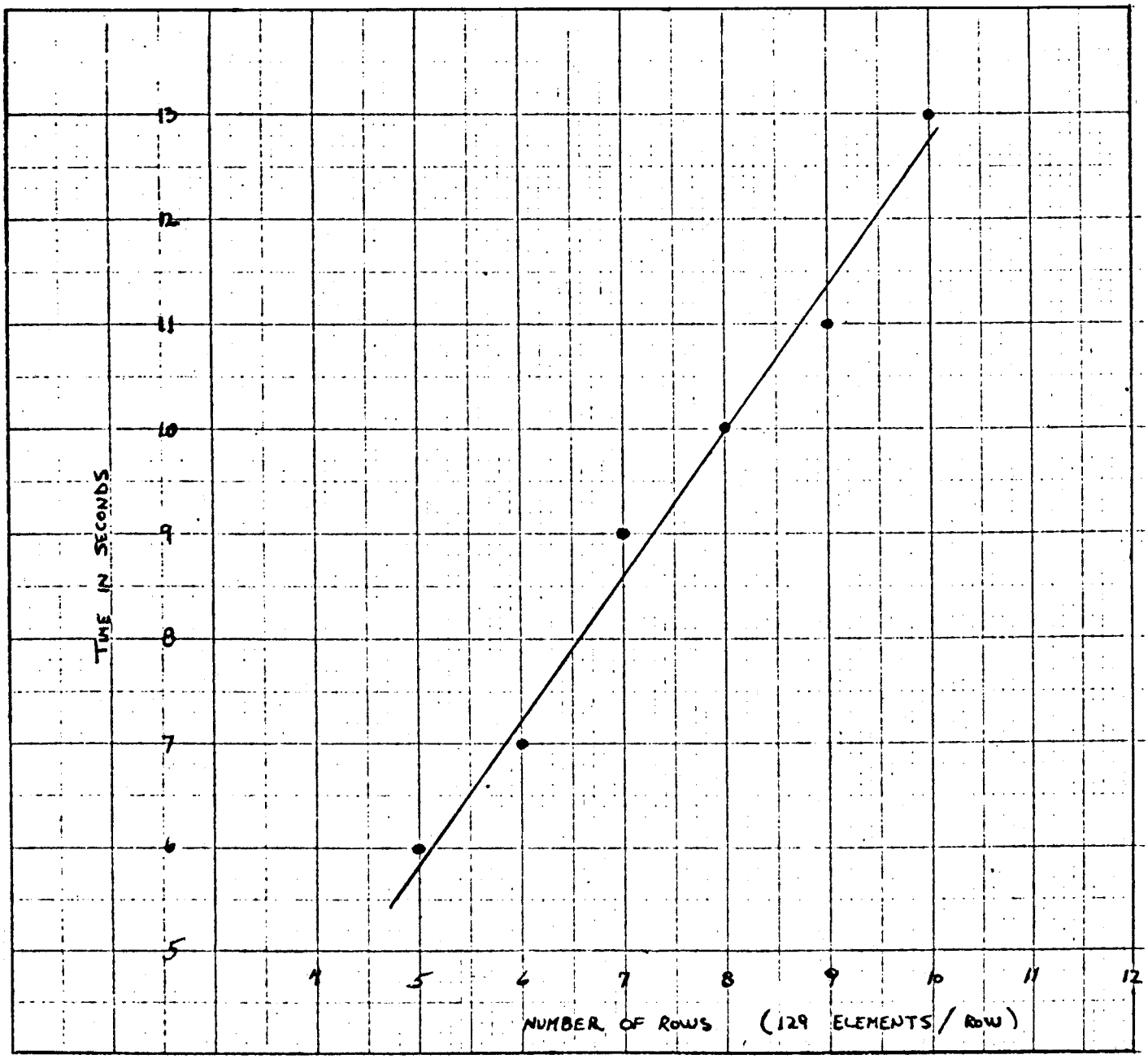


Figure B-2. MAT SORT/MAT MERGE Timing

APPENDIX C: SOME USEFUL DEFINITIONS

array: a number of related mathematical elements arranged in a specified order. Arrays can be one- or two-dimensional in the world of your Wang system. 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 of 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: Wang HEX, CRT Character Set and VAL Cross Reference 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

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.

700-3559E

TITLE OF MANUAL: **SORT STATEMENT REFERENCE MANUAL**

COMMENTS:

Fold

Fold

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

WANG

Fold

FIRST CLASS
PERMIT NO. 16
Tewksbury, Mass.

BUSINESS REPLY MAIL
NO POSTAGE STAMP NECESSARY IF MAILED IN THE UNITED STATES

– POSTAGE WILL BE PAID BY –

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

WANG LABORATORIES (CANADA) LTD.
49 Valleybrook Drive
Don Mills, Ontario M3B 2S6
TELEPHONE (416) 449-2175
Telex: 069-66546

WANG EUROPE S.A./N.V.
(European Headquarters)
250, Avenue Louise
1050 Brussels, Belgium
TELEPHONE 02/640.37.80
Telex: 12430-12398

WANG EUROPE S.A./N.V.
(Belgian Sales)
350, Avenue Louise
1050 Brussels, Belgium
TELEPHONE 02/648.91.00
Telex: 62691

WANG DO BRASIL COMPUTADORES LTDA.
Praca Olavo Bilac No. 28
SL1801/1803
Rio de Janeiro, Centro, RJ, Brasil
TELEPHONE 232-7503, 232-7026

WANG COMPUTERS (SO. AFRICA) PTY. LTD.
Corner of Allen Rd. & Garden St.
Bordeaux, Transvaal
Republic of South Africa
TELEPHONE (011) 48-6123
Telex: 960-83297

WANG INTERNATIONAL TRADE, INC.
One Industrial Avenue
Lowell, Massachusetts 01851
TELEPHONE (617) 851-4111
Telex: 94-7421

WANG SKANDINAVISKA AB
Pyramidvaegen 9A
S-171 36 Solna, Sweden
TELEPHONE 08/27 27 98
Telex: 11498

WANG COMPUTER LTD.
Shindaiso Building No. 5
2-10-7 Dogenzaka Shibuya-Ku
Tokyo, Japan
TELEPHONE (03) 464-0644
Telex: 2424909 WCL TKO J

WANG NEDERLAND B.V.
Produktieweg 1
Ijsselstein, Netherlands
TELEPHONE (03408) 41.84
Telex: 47579

WANG PACIFIC LTD.
9th Floor, Lap Heng House
47-50, Gloucester Road
Hong Kong
TELEPHONE 5-274641
Telex: 74879 Wang HX

WANG INDUSTRIAL CO., LTD.
7, Tun Hwa South Road
Sun Start Tun Hwa Bldg.
Taipei, Taiwan, China
TELEPHONE 7522068, 7814181-3
Telex: 21713

WANG GESELLSCHAFT MBH
Murlingengasse 7
A-1120 Vienna, Austria
TELEPHONE 85.85.33
Telex: 74640 Wang a

WANG GESELLSCHAFT MBH
Wiedner Hauptstrasse 68
A-1040 Vienna, Austria
TELEPHONE 57.94.20
Telex: 76424 Wang a

WANG S.A./A.G.
Markusstrasse 20
Postfach 423
CH 8042 Zurich 6, Switzerland
TELEPHONE 41-1-60 50 20
Telex: 59151

WANG COMPUTER PTY. LTD.
55 Herbert Street
St. Leonards, 2065, Australia
TELEPHONE 439-3511
Telex: 24569

WANG ELECTRONICS LTD.
Argyle House, 3rd Floor
Joel Street
Northwood Hills
Middlesex, HA6 INS, England
TELEPHONE (09274) 28211
Telex: 923498

WANG FRANCE S.A.R.L.
Tour Gallieni, 1
78/80 Ave. Gallieni
93170 Bagnolet, France
TELEPHONE 33.1.3602211
Telex: 680958F

WANG LABORATORIES GmbH
Moselstrasse 4
6000 Frankfurt AM Main
Postfach 16826
West Germany
TELEPHONE (0611) 252061
Telex: 04-16246

WANG DE PANAMA (CPEC) S.A.
Apartado 6425
Calle 45E, No. 9N. Bella Vista
Panama 5, Panama
TELEPHONE 69-0855, 69-0857
Telex: 3282243

WANG COMPUTER LTD.
302 Great North Road
Grey Lynn, Auckland
New Zealand
TELEPHONE Auckland 762-219
Telex: CAPENG 2826

WANG COMPUTER PTE., LTD.
Suite 1801-1808, 18th Floor
Tunas Building, 114 Anson Road
Singapore 2, Republic of Singapore
TELEPHONE 2218044, 45, 46
Telex: RS 24160 WANGSIN

WANG COMPUTER SERVICES
One Industrial Avenue
Lowell, Massachusetts 01851
TELEPHONE (617) 851-4111
TWX 710-343-6769
Telex: 94-7421

DATA CENTER DIVISION
20 South Avenue
Burlington, Massachusetts 01803
TELEPHONE (617) 272-8550

WANG LABORATORIES, INC.

ONE INDUSTRIAL AVENUE, LOWELL, MASSACHUSETTS 01851, TEL. (617) 851-4111, TWX 710 343-6769, TELEX 94-7421 Printed in U.S.A.
700-3559E
7-77-2.5M