MICRO B +™ PROGRAMMER'S GUIDE
ASSEMBLY LANGUAGE VERSION
FOR USE WITH LINK-80

SECOND EDITION
FEBRUARY 1981

FairCom
2606 Johnson Drive
Columbia, MO 65201
[314] 445-3304

MICRO B+™ PROGRAMMER'S GUIDE: LINK-80

Disclaimer

FairCom makes no representations or warranties with respect to the contents hereof and specifically disclaims any implied warranties of merchantability or fitness for any particular purpose. Further, FairCom reserves the right to revise this publication and to make changes from time to time in the content hereof without obligation of FairCom to notify any such person of such revision or changes.

MICRO B+™ PROGRAMMER'S GUIDE: LINK-80

TABLE OF CONTENTS

1.0 INTRODUCTION

This manual describes the use of MICRO B+™: a set of general purpose file accessing routines written in 8080 assembly language. To use this version of MICRO B+™, you must have a microcomputer system running under the CP/M[1] operating system, and MICROSOFT's LINK-80 linking loader or some other compatible loader. System development with MICRO B+™ will require a computer with at least 48K bytes of memory. However, applications including MICRO B+™ can run in substantially less memory since the code only requires between 4K and 8K bytes of memory (exclusive of buffer areas).

The design of MICRO B+™ is based on the B-Tree Index, one of the most powerful file accessing methods available today. Many mainframe computer manufacturers base their file accessing on the B-Tree approach. IBM's VSAM is one example. MICRO B+™ brings the full power of this approach to microcomputers. MICRO B+™ allows the application programmer to efficiently and easily maintain single or multiple key indices for large data files.

In general, a key index allows a user or programmer to locate a data record in a data file - without an exhaustive search of the file - just by knowing the key value associated with the desired data record. Multiple key indices allow a data record to be located on the basis of anyone of several key values. For example, a customer data file may use customer number and customer last name as keys. A particular customer's record can be located either by name or number; without the need to perform a lengthy search of all the customer records.

A key index method which also allows data records to be rapidly accessed in sequential (key value) order eliminates the need to perform time consuming, inconvenient sorts. Rapid key value access and rapid sequential access permit the design of powerful, on-line, interactive systems.

MICRO B+™ not only builds and maintains such key indices, it does so in a manner which minimizes the disk accesses necessary to find a key value in the index.

Please note that it is IMPERATIVE that back-ups of data files and index files be maintained no matter whose file accessing routines are utilized, and no matter how carefully you have conceived your application programs. Back-ups of data and indices are the only way to protect yourself, or users of your software from serious loss of a database.

---

[1] CP/M is a trademark of DIGITAL RESEARCH

2.0 SPECIAL FEATURES OF MICRO B+™

There are six special features which combined make MICRO B+™ the state-of-the-art in file accessing routines for microcomputers.

2.1 BALANCED TREE

Consistent with the B-Tree philosophy, MICRO B+™ ensures that the path length necessary to locate a key value in the index tree is the same for all key values. This is accomplished by maintaining a "balanced tree" (see Section 3), no matter how often the index is updated. The balanced nature of the index tree ELIMINATES the need to REORGANIZE the index.

2.2 FAST SEQUENTIAL ACCESS

Many file accessing mechanisms sacrifice the speed or ability to perform sequential key value accessing in order to provide rapid random access. Hash coding is a prime example of a key driven file accessing technique which does not permit efficient sequential processing. The B-Tree method provides sequential links as well as a balanced index search tree to achieve both rapid sequential processing and random accessing.

2.3 VIRTUAL DISK ACCESSING

Even though the basic B-Tree approach ensures a relatively small number of disk accesses to locate a particular key value, MICRO B+™ reduces the number of actual disk accesses even further by dynamically assigning frequently used B-Tree nodes to special I/O buffers. Therefore, locating a key value in the index may be accomplished without any actual disk accesses on some occaisions, and almost always with fewer disk accesses than would be required without the buffers.

2.4 DATA RECORD CONTROL

To ensure maximum flexibility for the application programmer, MICRO B+™ allows the programmer to control the actual data records. Data file support routines are provided with MICRO B+™ to help manage data records if desired. But the use of these data file routines is not required. The separation of index and data records permits the application programmer to use any data record formats as well as the most appropriate read and write routines to perform the actual data record I/O.

2.5 APPLICATION PROGRAM INTEGRATION

MICRO B+™ is designed to be fully integrated into your application programs. Mastering a small number of subroutine calls is all that is required to use MICRO B+™.

3.0 MICRO B+™ BASIC APPROACH

Managing a database requires two important activities:

- maintaining a "directory" which locates records in the database according to key values; and
- organizing the data records in the database (e.g., pile file, threaded list, stack, etc.).

MICRO B+™ addresses the first of these requirements, and gives the user complete freedom of choice on how to organize the data records. Data file routines are provided with MICRO B+™ which maintain data records according to a stack structure. These are provided as a convenience to the user; they are not required for the use of MICRO B+™.

There are two main approaches to realize the "directory" of key values:

- numerical transformation of the key value to a data record location (e.g., hash coding); and
- index tree search.

Subject to certain qualifications concerning the extent of overflow conditions, numerical key value transformation approaches are very fast. However, the key transformation approaches do not support sequential processing, nor do they support multiple keys. The index search tree methods achieve their efficiency for random access by partitioning the search for key values over successively smaller ranges of key values.

For the following discussion of the index tree search, refer to Figure 3-1. To locate a record with key value of 50, we would perform the following operations:

- access the root node;
- since 50 is greater than or equal to the index value of 43 in the root node, follow the right hand branch and access node 3;
- since 50 is greater than 48 and less than 60, follow the middle branch and access node 5; and
- since a match is found in leaf node 5, the data record with key value 50 is record number 1 in the data file.

Figure 3-2 shows a MICRO B+™ tree for the same information used in Figure 3-1 under the assumption of a maximum of four key values per node. Two features of this tree are noteworthy:

- The tree is balanced. This means that the number of nodes necessary to find a key value is the same for all key values. In the example, two node accesses are required for each of the nine key values.

- The leaf nodes are linked so that sequential access (in key value order) to the data records can be accomplished without traversing the tree.

The virtual disk accesses refered to in Section 2.3 are implemented by storing the information in a MICRO B+™ node in special I/O buffers. When a node is accessed by a MICRO B+™ routine, the I/O buffers are checked to see if the node is present. If present, no physical disk access is performed. If not present, the required node is read from disk into a dynamically assigned buffer. Once in a buffer, a node can be accessed and updated very rapidly. By increasing the number of I/O buffers, the number of actual disk accesses necessary to search the MICRO B+™ index tree can be dramatically reduced to achieve very fast access to anyone of a large number of data records.

The real power of the MICRO B+™ approach cannot be appreciated from Figure 3-2. Instead of allowing only four key values per node, consider what would happen if up to twenty key values are stored in each node. In this case, three node accesses (which may be performed with even fewer actual disk accesses because of the node I/O buffers) provide random access to the location of as many as 8,820 data records. Increasing the node size to accomodate up to forty key values means that three node accesses can locate the address of up to 67,240 records. (Note that these maximum capacities are not always attainable, but in no case would more than one additional node access be required to accomodate these data file sizes.)

This version of MICRO B+™ limits the number of key values stored in a node to one-hundred twenty-four (124).

For a more complete description of the B-Tree approach, please refer to

Comer, D., "The Ubiquitous B-Tree," **ACM COMPUTING SURVEYS**, Volume 11, Number 2, June 1979 (pp. 121-137).

Knuth, D.E.,**The Art Of Computer Programming Volume 3: Sorting And Searching**, Addison-Wesley Publishing Company, Reading, Massachusetts, 1973 (pp. 451-479).

4.0 OVERVIEW OF MICRO B+™ ROUTINES

This Section gives a brief overview of the MICRO B+™ routines which are callable either from other assembly language routines or high level languages, such as MICROSOFT Basic, which support calls to assembly language routines. Sections 4.2 through 4.14 provide more complete documentation of the routines.

ROUTINE NAME	PARAMETERS (I=INPUT O=OUTPUT)	PURPOSE
INTREE*	1. Number of Buffers(I) 2. Number of Keys(I) 3. Number of 128 byte sectors per node(I) 4. Number of 128 byte sectors in the file header record(I) 5. Number of data files supported by the assembly language routines of Section 6.(I)	To initialize the MICRO B+™ routines and setup the buffer area.
SETERR	1. Error Code(I/O)	To establish a variable which is set to a non-zero value if a user error occurs. If no call is made to SETERR, user errors will cause an error message followed by a warm reboot.
ACCESS*	1. Key Number(I) 2. Index File Name(I)* 3. Key Length (bytes)(I) 4. Key Type (0-alpha,1-numeric)(I) 5. Maximum number of key values per node(I)	To open or create an index file.
ENTER*	1. Key Number(I) 2. Key Value(I)* 3. Data Record Number(I) 4. Return Code(O) 1-successful 2-duplicate key value 3-key not ACCESSed	To add a new Key Value to the specified B-Tree Index. The associated data record number is stored with the Key Value in the index.

ROUTINE NAME	PARAMETERS (I=INPUT O=OUTPUT)	PURPOSE
RTRIEV*	1. Key Number(I) 2. Key Value(I)* 3. Data Record Number(O)	To return the data record number associated with the key value. Zero is returned if the key is not found.
SEARCH*	1. Key Number(I) 2. Key Value(I)* 3. Data Record Number(O) 4. Index Key(O)*	To find the first index entry which is geater than or equal to the specified Key Value. The data record number associated with the index entry is returned and Index Key is set equal to the index entry. If the search is unsuccessful, a zero is returned for the data record number.
SUCESR	1. Key Number(I) 2. Data Record Number(O) 3. Index Key(O)*	To find the next entry in the specified index. The associated data record number is returned and Index Key is set equal to the index entry. If there is no "next" entry, a zero is returned for the data record number.
PRDESR	1. Key Number(I) 2. Data Record Number(O) 3. Index Key(O)*	To find the previous entry in the specified index. The associated data record number is returned and Index Key is set equal to the index entry. If there is no previous entry, a zero is returned for the data record number.
NMENTR	1. Key Number(I) 2. Number of Entries(O)	To determine the number of entries in the specified index.
NMNODE	1. Key Number(I) 2. Number of Nodes(O)	To determine the number of nodes in the specified index.

ROUTINE NAME	PARAMETERS (I=INPUT O=OUTPUT)	PURPOSE
REMOVE *	1. Key Number(I) 2. Key Value(I)* 3. Data Record Number(IO) 4. Return Code(O) 0-Key Value not found 1-successful 2-Data Record Number does not agree with associated number in the index 3-Key not ACCESSed	To delete an index entry from the specified index. If the Data Record Number supplied is zero, it will be set to associated data record number stored in the index for Key Value.
RSTRCT *	1. Key Number(I)	To close the specified index file.
INTLOD	1. Key Number(I) 2. Load Factor(I)	To initialize MICRO B+™ for the construction of a new index file with key values presented in strictly increasing order.
LOADKY	1. Key Value(I)* 2. Data Record(I) 3. Return Code(O) 1-successful 2-duplicate key value 3-key value out of order	To add Key Values, in sequential order, to B-Tree leaf nodes.
BLDIND		To build the upper levels of the B-Tree after all calls to LOADKY have been made.

NOTE: Parameters followed by an asterisk (*) are STRING entities. All other parameters are sixteen (16) bit integer quantities.

NOTE: Key Numbers must be in the range from zero (0) to the Number of Keys less one (1); where Number of Keys is the second parameter in the INTREE routine.

4.1 PARAMETER PASSING PROTOCOL

Parameters are passed to the assembly language routines according to the MICROSOFT "CALLing" standard:

- Return Address is placed on the Stack;
- Address of 1st parameter is in HL register pair;
- Address of 2nd parameter is in DE register pair;
- If only three parameters are required, address of 3rd parameter is in BC register pair;
- If four or more parameters are required, BC contains the address of a list of parameter addresses: the address of the 3rd parameter is in the first two bytes of the list, the address of the 4th parameter is in the next two bytes of the list, etc.;

If your host language does not use this parameter protocol, it will be necessary to code interface routines to satisfy the above standard.

4.1.1 PARAMETER SPECIFICATIONS

Two types of parameters are used in the MICRO B+™ routines: string-valued quantities and two-byte integers. The parameters followed by an asterisk (*) in Section 4.0 (viz., Key Value, Index Key, and Index File Name) are string-valued quantities (even if the Key Type designates numeric Key Values). All other parameters are two-byte integer quantities.

All two-byte integer quantities are assumed to be stored in memory with the Least Significant Byte first. Therefore, when the address of a such a parameter is passed to a MICRO B+™ routine, the address points to the Least Significant Byte of the parameter.

The address passed to MICRO B+™ routines for string-valued parameters must point to the first byte of a three-byte vector formed as follows:

1st byte: length of string quantity in bytes;
2nd byte: Least Significant Byte of the actual address of the string;
3rd byte: Most Significant Byte of the actual address of the string.

For example, if the string expression

"I AM A STRING"

is stored beginning at memory location 501EH, then the three-byte vector describing this string is

0D1E50H

Please note that if this string expression were to be passed as a parameter, then the address passed should point to the first byte of the three-byte vector (which in turn points to the actual string).

4.1.2 INTERFACE PROGRAMMING EXAMPLE

For this example, assume that the host language passes parameters according to the following protocol:

- return address is placed on the stack;
- address of the list of argument addresses is in the register pair HL; and
- the address of a three-byte vector as described above is passed for string valued parameters.

Further assume that it is desired to ENTER a new key value into a B-Tree index. In this case, the following 8080 assembly language code could be used to interface with the MICRO B+™ routine:

```
        EXT     ENTER   ;DECLARE ENTER AS AN EXTERNAL NAME (You
                        ; must have an assembler which can create
                        ; relocatable object modules and allow for
                        ; the declaration of external names and
                        ; entry points.)
        ENTRY   ADDKEY  ;DECLARE ADDKEY AS AN ENTRY POINT (ADDKEY is
                        ; the routine name you would use to
                        ; perform the MICRO B+™ ENTER routine.)
ADDKEY: MOV     E,M     ;HL = ADDRESS OF 1ST ENTRY IN ARGUMENT
                        ; ADDRESS LIST
        INX     H
        MOV     D,M     ;DE = ADDRESS OF 1ST PARAMETER
        PUSH    D       ;SAVE ADDRESS OF 1ST PARAMETER
        INX     H       ;HL = ADDRESS OF 2ND ENTRY ON LIST
        MOV     E,M
        INX     H
        MOV     D,M     ;DE = ADDRESS OF 2ND PARAMETER
        INX     H       ;HL = ADDRESS OF 3RD ENTRY ON LIST
        MOV     B,H
        MOV     C,L     ;BC = ADDRESS OF ARGUMENT LIST STARTING
                        ; WITH THE 3RD PARAMETER
```

HL = 1

DE = 2

BC = 3 or param list

4.2 INTREE(NO.BUFFERS%,NO.KEYS%,NO.NODE.SECTORS%,NO.HEADER.SECTORS%, NO.DATA.FILES%)

[In order to provide a concise specification of the parameters required by each MICRO B+™ routine, the remaining sections of Chapter 4 will use MICROSOFT Basic Version 5 as a model: identifiers ending in "%" are two-byte integers, those ending in "$" are string-valued quatities, those ending in "!" are four-byte floating point quantities, and identifiers ending in "#" are eight-byte floating point quantities.]

Before any index files (please note that the terms "index file" and "key file" will be used interchangeably in this manual, and that they refer to the disk file which contains the actual B-Tree index) can be opened or used, the routine INTREE, which initializes the special MICRO B+™ buffer area and specifies the basic characteristics of the index files, must be executed.

NO.BUFFERS% specifies the number of index file I/O buffers that will be used. There must be at least three (3) buffers and no more than eighteen (18) buffers. As the number of buffers is increased, the time to access a key value decreases while the memory space required increases. Note that all the index files share the same buffer space, thereby minimizing the memory required to implement applications. Even if ten (10) index files are used simultaneously, it is not necessary (although it may be desirable) to use more than three (3) buffers.

NO.KEYS% specifies the maximum number of index files that may be accessed simultaneously. NO.KEYS% must be at least one (1) and no more than ten (10). This limit may be extended in custom versions; please contact FairCom for details.

NO.NODE.SECTORS% determines the record length of the index files. Specifically, NO.NODE.SECTORS% is the number of 128-byte disk sectors in each index file record. Each index file record corresponds to a B-Tree node (as illustrated in Figure 3-2). The more sectors per record, the more key values stored per node. The more key values stored per node, the less node accesses required to find a key value. NO.NODE.SECTORS% must be at least one (1). There is no upper bound on this parameter, but since the maximum number of key values stored per node is one-hundred twenty-four (124) there are some practical limitations. It is very important to note that all index files which are used simultaneously are forced to have the same record length.

NO.HEADER.SECTORS% specifies the number of 128 byte disk sectors used for the header record (which is the first record) of the index file. The header record maintains status information about the size of the index file and the root of the B-Tree.

Although this information only requires nine (9) bytes, it is not always advisable to choose a value of one (1) for NO.HEADER.SECTORS%. NO.HEADER.SECTORS% should be set to the number of 128 byte sectors in a physical block for your disk drive.

For example, if you are using a double density disk drive (which transfers 256-byte blocks of data), and you have set NO.NODE.SECTORS% to two (2), then chosing a value of one (1) for NO.HEADER.SECTORS% would degrade your response time since each B-Tree node would be split between two physical blocks on your disk, and would require two disk accesses for each node read and write operation. Choosing a value of two (2) for NO.HEADER.SECTORS% would allow each node to be read or written with only one disk access.

NO.DATA.FILES% specifies the maximum number of data files to be opened simultaneously using the routines described in Section 6 of this guide. (Section 6 describes assembly language routines to support data file I/O.) PLEASE NOTE THAT IT IS NOT NECESSARY TO USE THE ROUTINES IN SECTION 6. YOU MAY USE THE ROUTINES DESCRIBED IN SECTION 5 (WHICH ARE WRITTEN IN BASIC SOURCE CODE), OR YOU MAY USE YOUR OWN DATA FILE ROUTINES. IF YOU DO NOT USE THE ROUTINES IN SECTION 6, SIMPLY SET NO.DATA.FILES% TO ZERO (0).

The maximum number of data files supported by the routines of Section 6 is twenty (20). This limit can be increased in a custom version of MICRO B+™.

Once INTREE has been called, additional calls to INTREE (say to set up a new set of index files) should only be made after all opened index (and Section 6 data) files have been closed.

4.2.1 BUFFER SIZE COMPUTATION

The special I/O buffers which are used to facilitate node accesses require the following amount of memory space:

```
((NO.KEYS%+ NO.DATA.FILES%) * 36) +
     (NO.BUFFERS% * ((NO.NODE.SECTORS% * 128) + 50))
```

Recall that this memory space is required in addition to that used by the MICRO B+™ routines themselves.

For example, three (3) keys, one (1) data file (supported by the routines of Section 6), ten (10) buffers, and a 256-byte index record length will require 3204 bytes of memory.

4.3 **SETERR**(ERROR.CODE%)

SETERR allows an application program to make an orderly recovery from User Errors. MICRO B+™ distinguishes between two types of errors: User Errors and MICRO B+™ Errors.

Ordinarily, MICRO B+™ Errors should never occur. They result from illegal conditions arising in the B-Trees. When a MICRO B+™ Error occurs, a message is sent to the console device, and a warm boot is executed. Please call FairCom at (314) 445-3304 if a MICRO B+™ error ever occurs and you cannot determine the cause. One possible cause is trying to build an index file greater than 512K bytes while operating under CP/M® 1.4.

User Errors occur when a problem condition arises which is otherwise avoidable by the application program. For example, trying to create a new index file on diskette which is full will result in a User Error. If SETERR is called, the application program can trap user errors and perform an appropriate action.

ERROR.CODE% serves as both an input and output parameter. If SETERR is called with a non-zero value for ERROR.CODE% (the particular non-zero value is of no significance), then each subsequent call to a MICRO B+™ routine will

- cause ERROR.CODE% to be set to zero if no User Error occurs; or

- cause ERROR.CODE% to be set to one of the non-zero User Error values described in Section 8.1.

In either case, control will be passed to the statement following the call to MICRO B+™. Hence, if the following statement tests the value of ERROR.CODE%, the application program can trap User Errors in order to make a meaningful recovery.

If SETERR is called with a zero value for ERROR.CODE%, or never called at all, then subsequent User Errors will print a message on the console device followed by a warm boot (back to CP/M®).

Note that SETERR can be called as often as desired, but that SETERR needs only to be called once unless it is desired to either change the ERROR.CODE% variable, or to change the error handling protocol.

A list of User Error codes is given in Section 8.1.

PLEASE NOTE THAT IF SETERR IS INVOKED (with a non-zero ERROR.CODE%) PRIOR TO THE USE OF THE ACCESS ROUTINE, IT IS IMPERATIVE THAT THE VALUE OF ERROR.CODE% BE TESTED AFTER EACH CALL TO ACCESS. IF SUCH TESTS ARE NOT PERFORMED, AND IF THE

INDEX WAS NOT PROPERLY ACCESSED, IT IS POSSIBLE FOR UNPREDICTABLE ERRORS TO OCCUR IN OTHER ROUTINES.

4.4 ACCESS(KEY.NO%,INDEX.FILE$,KEY.LENGTH%,KEY.TYPE%,MAX.NO.KEYS%)

ACCESS is used to open or create the INDEX.FILE$ and assign KEY.NO% to this file. All subsequent references to INDEX.FILE$ are made via the KEY.NO% value.

KEY.NO% must be in the range from zero to NO.KEYS%-1.

KEY.LENGTH% specifies the maximum length (in bytes) for key values. KEY.LENGTH% must be at least one (1) and not greater than forty-eight (48). Numeric keys must be at least two (2) bytes in length.

KEY.TYPE% specifies whether the key values are to be stored in lexicographically increasing order (i.e., as alphanumerics), or in numerical order. KEY.TYPE% equal to zero (0) indicates alphanumeric key values while a value of one (1) indicates numeric key values. It is especially important to note that all key values are passed to and from the MICRO B+™ routines according to the protocol for string-valued quantities even if the key is designated as a numeric key. The KEY.TYPE% designation only affects the order in which the keys are stored.

Numeric key values are assumed to be integers (not, however, restricted to only two bytes) stored with the Least Significant Byte first and the Most Significant Byte (which includes the sign of the integer) last. This format is automatically generated in MICROSOFT Basic for two-byte integers using the MKI$ intrinsic function.

MAX.NO.KEYS% determines the maximum number of key values stored in each B-Tree node (and, hence, in each index file record). Recall that as MAX.NO.KEYS% is increased, the number of levels in the index structure decreases. The number of levels in the index structure determines the node accesses required to locate a key value. (Please note that a node access causes a disk access only when the node is not in one of the special MICRO B+™ I/O buffers.)

MAX.NO.KEYS% must be an EVEN integer in the range from two (2) to one-hundred twenty-four (124). Further, MAX.NO.KEYS% must be less than or equal to

```
((NO.NODE.SECTORS% * 128) - 8) / (KEY.LENGTH% + 2)
```

4.4.1 MULTIPLE INDEX FILES

When more than one index file is to be ACCESSed simultaneously, the selection of NO.NODE.SECTORS% and MAX.NO.KEYS% must take into account your desire for

speedy response times and efficient disk storage. While large values for MAX.NO.KEYS% lead to fast response times, it may not be advisable to set this parameter the same for all INDEX.FILE$'s. If the KEY.LENGTH%'s are not the same for each file, there may be wasted disk space if MAX.NO.KEYS% is set the same for each file since NO.NODE.SECTORS% must be set to accomodate the longest KEY.LENGTH%.

The selection of NO.NODE.SECTORS%, and each of the MAX.NO.KEYS%'s parameters should reflect the relative importance of response time and disk storage space. Note also, that NO.BUFFERS% can be used to improve response time (at the expense of memory utilization) without increased disk space utilization. See Section 7.2 for additional discussion of the buffer area.

4.4.2 ESTIMATING INDEX FILE SIZE

Once you have chosen the values for NO.NODE.SECTORS%, NO.HEADER.SECTORS%, and MAX.NO.KEYS%, the minimum index file size possible for a given number of index ENTRIES% is computed as follows:

```
NODES% = <ENTRIES% / MAX.NO.KEYS%> [where <X> is the
         smallest integer greater than or equal to X]
INDEX.NODES% = NODES%
WHILE INDEX.NODES% > 1
     INDEX.NODES% = <INDEX.NODES% / (MAX.NO.KEYS% + 1)>
     NODES% = NODES% + INDEX.NODES%
WEND
INDEX.FILE.SIZE% = NODES% * NO.NODE.SECTORS% * 128 +
     NO.HEADER.SECTORS% * 128
```

To compute the largest index file size possible, replace MAX.NO.KEYS% by MAX.NO.KEYS%/2 in the above algorithm. The minimum size computation assumes completely full nodes while the largest size computation assumes half-full nodes. The ordinary B-Tree structure ensures at least half-full nodes. MICRO B+™ performs "local node rotations" to help maintain the smallest index file size by avoiding unnecessary node splitting.

For example, using the above algorithm, it will require between 87K and 180K bytes to store 10,050 index entries in an index file for which

```
NO.NODE.SECTORS%=2
NO.HEADER.SECTORS%=1
MAX.NO.KEYS%=30
```

These values correspond to the bench mark speed test

which FairCom has performed for MICRO B+™. In that test, in which the key values were generated totally at random, the actual index file size was 118K bytes which implies that the nodes were approximately 75% full.

4.4.3 MICROSOFT BASIC EXAMPLE

```
10 BUFS%=10:KEYS%=2:NODE.SEC%=2:HEAD.SEC%=1:DAT.FILE%=0
20 CALL INTREE(BUFS%,KEYS%,NODE.SEC%,HEAD.SEC%,DAT.FILE%)
30 USER.ERROR.NO%=1:CALL SETERR(USER.ERROR.NO%)
40 CUST.KEY%=0:PART.KEY%=1
50 CUST.LEN%=11:CUST.TYPE%=0:CUST.VAL%=18
60 PART.LEN%=2 :PART.TYPE%=1:PART.VAL%=62
70 CALL ACCESS(CUST.KEY%,"B:CUST.IND",CUST.LEN%,
     CUST.TYPE%,CUST.VAL%)
80 IF USER.ERROR.NO%<>0 THEN GOSUB 9000
90 CALL ACCESS(PART.KEY%,"C:PART.IND",PART.LEN%,
     PART.TYPE%,PART.VAL%)
100 IF USER.ERROR.NO%<>0 THEN GOSUB 9000
```

In this example, there will be ten (10) I/O buffers set up, a maximum of two (2) key files ACCESSible at one time, an index file CUST.IND will be opened (or created) on drive B which accomodates up to eighteen (18) eleven-byte alphanumeric key values per node, and an index file PART.IND will be opened on drive C which accomodates up to sixty-two (62) two-byte numeric keys per node. If a User Error occurs during the ACCESS operations, control will be transferred to a subroutine where the desired action will be taken.

4.4.4 CP/M® 1.4 AND 2.x COMPATIBILITY

ACCESS automatically determines which version of CP/M® is resident, and uses the appropriate disk I/O facilities. Two routines - ACCES1 and ACCES2 - can be used in place of ACCESS to coerce MICRO B+™ to use the disk I/O facilities of CP/M version 1 and 2, respectively. Both ACCES1 and ACCES2 require the same parameters as ACCESS.

UNPREDICTABLE, AND ALMOST CERTAINLY BAD, ACTIONS WILL OCCUR IF A PROGRAM USES BOTH ACCES1 AND ACCES2.

4.5 ENTER(KEY.NO%,KEY.VALUE$,DATA.RECORD%,RETURN.CODE%)

ENTER adds KEY.VALUE$ to the index file specified by KEY.NO% and associates the DATA.RECORD% number with the KEY.VALUE$. It is the programmer's responsibility to ensure that the DATA.RECORD% does correspond to the KEY.VALUE$.

RETURN.CODE% is set according to the following:

1 if the KEY.VALUE$ insertion is successful;
2 if the KEY.VALUE$ is already in the index; and
3 if the index file specified by KEY.NO% has not been ACCESSed.

If KEY.VALUE$ is a null string, then no action is taken by ENTER; but RETURN.CODE% is set to one (1). A string is null if the first byte of the three-byte vector (see Section 4.1.1) defining the string is zero.

4.5.1 EXTRA LARGE FILES

While the standard version of MICRO B+™ is capable of handling index files with up to 65,535 entries, the index file itself must reside in one file. In fact, the only actual physical limitation on the size of the index file is determined by the largest file supported by CP/M® on your system. If more than 65,535 entries are added to the index, the consequences are:

- the internal counter which tracks the number of index entries will be reset to zero (i.e., turn over); and
- the two-byte associated data record numbers may not be sufficient to address all the data records. In this case, one byte can be appended to the end of the key values to determine the actual associated data record number.

The data files to which the index files maintain pointers need not be restricted to one logical and/or physical file. Since the index file and data file are separate entities, it is possible to allow the data file to be spread over more than one disk file. Such segmentation of the data file can be represented in the index file either by how the associated DATA.RECORD% number is interpreted, or by appending an appropriate code to the KEY.VALUE$ itself.

For example, one may assume that a data base will be split into files of 1000 records each. Then, the associated DATA.RECORD% number is interpreted both as indicator of data file as well as position within the data file (e.g., DATA.RECORD%=9438 implies the 438th record of the 9th data file).

4.5.2 CODING NUMERIC KEY VALUES

As mentioned several times already, all KEY.VALUE$'s are passed to the MICRO B+™ routines as string-valued quantities; even when the KEY.TYPE% indicates a numeric key. If the KEY.LENGTH% is two (2) for a numeric key, then to transform an integer variable to the appropriate string representation in MICROSOFT Basic, one need only invoke the MKI$ intrinsic function. However, if your host language does not have such a conversion function, or if the KEY.LENGTH% is greater than two (2), then the following algorithm can be used to create a string representation of a numeric (actually integer) key. Please note that this algorithm assumes a positive integer is to be converted; if the number is not positive, then it should be converted to a positive by adding it to the appropriate upper bound which depends on the KEY.LENGTH% (e.g., 256^2=65536 in the case of two byte integers). The algorithm also assumes the existence of a function, CHR$, to convert integer values in the range from 0 to 255 to a string quantity.

```
KEY.VALUE$=""
FOR B%=1 TO KEY.LENGTH%
        FACTOR=INT(NUMBER/256.)
        BYTE%=NUMBER-256.*FACTOR
        NUMBER=FACTOR
        KEY.VALUE$=KEY.VALUE$+CHR$(BYTE%)
NEXT B%
```

where NUMBER is initially set to the value to be converted and INT(X) is the largest integer less than or equal to X. This algorithm assumes that NUMBER can be represented in KEY.LENGTH% bytes.

It is also possible to convert a number to its ASCII representation (say with a function such as STR$ in MICROSOFT Basic). However, if this is done, then the KEY.TYPE% must indicate an alphanumeric key (i.e., must be zero (0)). Further, the KEY.VALUES$'s will not be stored in numeric order unless a very careful right-justification scheme is employed; and the space savings which are gained by converting from an ASCII to binary representation will not be realized.

4.5.3 KEY VALUE PADDING

ENTER pads KEY.VALUE$'s that are less than KEY.LENGTH% bytes with blanks (20H) on the right, and truncates KEY.VALUE$'s that are too long. However, to ensure proper handling of numeric keys, it essential that all numeric KEY.VALUE$'s are passed to the MICRO B+™ routines with the exact KEY.LENGTH%.

There may also be situations where it is desirable to store alphanumeric KEY.VALUE$'s in right-justified form. If so, it is the programmer's responsibility to ensure that the KEY.VALUE$'s have been properly justified in view of the truncation of "too long" keys.

4.5.4 DUPLICATE KEY VALUES

There are situations, such as building an index based on last names, where the key values are not inherently unique. When this arises, it is necessary to append a unique identifier to the key, possibly after truncating the original key value to a prespecified length. In this manner, the KEY.VALUE$'s will still be stored in the expected order, but there will not be any conflicts between like-valued entries.

The easiest identifier to append in this situation is the record number in the data file which corresponds to the KEY.VALUE$. In cases where there is, in addition to the potentially non-unique key, a unique key associated with each database entry, an alternative is to append the unique key to the end of the non-unique key value since this creates an automatic cross reference capability.

4.5.5 MICROSOFT BASIC EXAMPLE

```
110 INPUT "ENTER NEW PART NUMBER:";PART.NO%
120 PART.NO$=MKI$(PART.NO%)
130 CALL ENTER(PART.KEY%,PART.NO$,NEW.REC%,CODE%)
140 IF USER.ERROR.NO%<>0 THEN GOSUB 9000
    ELSE IF CODE%<>1 THEN GOSUB 8000
```

In this example, the key value PART.NO$ will be added to an index file provided that the file has already been ACCESSed and that such an entry is not already in the index file. Either of these conditions will be reflected in the value assigned to CODE%. It has been assumed that the record number in the data base corresponding to PART.NO$ has already been assigned to the variable NEW.REC%. Further, it has also been assumed that SETERR was previously called with USER.ERROR.NO% as the argument.

4.6 RTRIEV(KEY.NO%,KEY.VALUE$,DATA.RECORD%)

RTRIEV returns the DATA.RECORD% number associated with the KEY.VALUE$ in the index specified by the KEY.NO%. If no such KEY.VALUE$ exists in the index file, then DATA.RECORD% is set to zero.

4.6.1 MICROSOFT BASIC EXAMPLE

```
150 INPUT "ENTER DESIRED PART #:";PART.NO%
160 PART.NO$=MKI$(PART.NO%)
170 CALL RTRIEV(PART.KEY%,PART.NO$,DATA.BASE.POINTER%)
180 IF  DATA.BASE.POINTER%=0  THEN
        PRINT "NO SUCH ENTRY IN DATA BASE!"
```

4.7 SEARCH(KEY.NO%,KEY.VALUE$,DATA.RECORD%,INDEX.KEY$)

SEARCH returns the DATA.RECORD% number associated with the first entry (in key-sequential order) in the index file which is equal to or greater than KEY.VALUE$. INDEX.KEY$ is set equal to this index entry. If no such index entry is found during the SEARCH, then DATA.RECORD% is set to zero and INDEX.KEY$ is set to all blanks (which is not a null string).

Two important notes concerning INDEX.KEY$ must be made. First, since MICRO B+™ is expected (but not required) to be used with host languages which dynamically manage a string storage space, MICRO B+™ never changes the contents of the three-byte vector (see section 4.1.1) which defines a string-valued quantity. Therefore, if INDEX.KEY$ is not initially set to a string value which is long enough to contain the string-valued index entries which will be assigned to INDEX.KEY$, then MICRO B+™ will be unable to make the assignment resulting in a User Error. Therefore, at the beginning of an application program, it is necessary to set up one or more string variables which can be used in subsequent calls to SEARCH, SUCESR, and PRDESR. The set-up can be simply assigning sufficiently long blank strings to these variables. Second, if INDEX.KEY$ is longer than the KEY.LENGTH% associated with KEY.NO%, then INDEX.KEY$ will be padded on the right with blanks.

4.7.1 ON DUPLICATE KEY VALUES

As mentioned in Section 4.5.4, situations arise where key values must be modified to accomodate duplicate key values. When this is done, it is no longer possible to use RTRIEV to locate such an index entry since the original value will have been modified. SEARCH can be used in this case to find the first candidate for a match with the desired key value. Subsequent candidates can be found by using SUCESR which is described below.

4.8 SUCESR(KEY.NO%,DATA.RECORD%,INDEX.KEY$)

SUCESR returns the DATA.RECORD% number associated with the next key value stored in the index specified by the KEY.NO%. INDEX.KEY$ is set equal to this "next" entry. If no next entry exists, then DATA.RECORD% is set to zero (0) and INDEX.KEY$ becomes a blank (not null) string.

Before the first call to SUCESR for a given KEY.NO%, either RTRIEV or SEARCH must have been called for the same KEY.NO%. Each time RTRIEV or SEARCH are called, the position in the index file is reset for subsequent calls to SUCESR. Each time SUCESR is called, the position pointer is advanced so that successive calls to SUCESR allow you to step through the index file in key-sequential order. Please note that since all KEY.VALUE$'s are stored in the leaves of the B-Trees, it is not necessary to traverse the tree to find the next entry.

Since separate position pointers are maintained for each index file, it is allowable to interleave calls to SUCESR for different KEY.NO%'s.

Please see Section 4.7 for a discussion of how INDEX.KEY$ must be initialized.

4.8.1 MICROSOFT BASIC EXAMPLE

```
180 FIELD #3,16 AS LAST.NAME$,12 AS FIRST.NAME$,
     4 AS ACCT.RECV$
190 ENTRY$=SPACE$(11) 'SET ENTRY$ TO THE PROPER LENGTH
200 INPUT "ENTER CUSTOMER LAST NAME:";CUST.NAME$
210 CUST.NAME$=LEFT$(CUST.NAME$+SPACE$(16),16) 'SET
     VARIABLE TO SAME LENGTH AS FIELDed VARIABLE
215 KEY$=LEFT$(CUST.NAME$,9) 'SET KEY$ TO LENGTH OF
     TRUNCATED KEY.VALUE$
216 SRCH.KEY$=KEY$+MKI$(0) 'APPEND BINARY ZERO TO END OF
     KEY SO THAT IT WILL NOT BE PADDED WITH BLANKS (20H)
220 CALL SEARCH(CUST.KEY%,SRCH.KEY$,REC.NO%,ENTRY$)
230 WHILE KEY$=LEFT$(ENTRY$,9) 'TEST FOR MATCH OF
                                TRUNCATED KEY VALUES
240   GET 3,REC.NO%
250   IF LAST.NAME$=CUST.NAME$ THEN
          PRINT FIRST.NAME$,LAST.NAME$,
               CVS(ACCT.RECV$) 'TEST FOR MATCH OF
                                COMPLETE NAME
260   CALL SUCESR(CUST.KEY%,REC.NO%,ENTRY$)
270 WEND
280 PRINT "SEARCH ENDED"
```

In this example, SEARCH is used to find the first potential match for the desired CUST.NAME$ (which is truncated to nine bytes and has a two-byte identifier added before it is ENTERed in the index file). Subsequent calls to SUCESR are used to find the

remaining potential matches. Note that the WHILE loop is used to step through the index until the truncated CUST.NAME$ no longer matches the truncated index entries. The SRCH.KEY$ is padded with binary zeroes to avoid padding with blanks to the full key length of eleven (11) bytes.

4.8.2 SEQUENTIAL PROCESSING AND INDEX UPDATES

Caution must be exercised if one desires to step through an index file in sequential order while performing occaisional additions and/or deletions to the index. While all key values are linked both forwards and backwards to their nearest neighbors, and while these links are always maintained during additions and/or deletions to the index, the actual sequential processing is controlled by internal pointers (which are not a part of the index file itself). These pointers determine the current position in the index, and are updated by calls to RTRIEV, SEARCH, SUCESR, and PRDESR. However, these internal pointers are not maintained during additions and/or deletions to the index.

Therefore, if you wish to sequentially traverse the index and at the same time perform aditions and/or deletions to the index, the following strategy should be adopted:

Step 1 - use the sequential access capability to step through the index file until an insertion (ENTER) or deletion (REMOVE) is performed.

Step 2 - after the insertion or deletion, use RTRIEV and/or SEARCH to re-establish your position in the index file. Then return to Step 1.

Section 9.2 contains an extensive example program which includes the use of SEARCH to reset the internal pointers. (See, in particular, line 5385 of the example.)

4.9 PRDESR(KEY.NO%,DATA.RECORD%,INDEX.KEY$)

PRDESR returns the DATA.RECORD% associated with the previous key value stored in the index specified by KEY.NO%. That is, PRDESR allows the application program to traverse the key values in reverse order. Otherwise, PRDESR behaves exactly like SUCESR. See the Section 4.8 for more details.

Please note that calls to SUCESR and PRDESR may be intermixed freely.

4.10 NMENTR(KEY.NO%,NO.ENTRIES%)

NMENTR sets the variable NO.ENTRIES% to the number of key values stored in the index file specified by KEY.NO%. The number of entries ranges from zero (0) to 65,535.

Note that if NO.ENTRIES% is greater than 32,767, it will be a negative number. Adding 65,536 to a negative NO.ENTRIES% results in the correct value.

If more than 65,535 entries are added to an index file, it will cause NO.ENTRIES% to start counting over from zero. It is the application program's responsibility to determine how to interpret the result of calling NMENTR.

4.10.1 MICROSOFT BASIC EXAMPLE

```
300 CALL NMENTR(PART.KEY%,NO.OF.PARTS%)
310 IF NO.OF.PARTS% < 0 THEN
        NO.OF.PARTS! = 65536! + NO.OF.PARTS%
    ELSE
        NO.OF.PARTS! = NO.OF.PARTS%
```

This example demonstrates how the size of an index file, measured in number of entries, can be easily determined; and subsequently transformed to a four-byte floating point representation.

4.11 NMNODE(KEY.NO%,NO.NODES%)

NMNODE sets the variable NO.NODES% to the number of B-Tree nodes in use and available for use in the index file specified by KEY.NO%.

4.11.1 MICROSOFT BASIC EXAMPLE

```
400 CALL NMNODE(PART.KEY%,NODES%)
410 TOTAL.SECTORS% = NO.HEADER.SECTORS% + NODES% *
        NO.NODE.SECTORS%
420 TOTAL.KBYTES% = (TOTAL.SECTORS% + 7) / 8
```

In this example, which assumes that NO.HEADER.SECTORS% and NO.NODE.SECTORS% were used in the call to INTREE (see Section 4.2), TOTAL.KBYTES% is set to the number of kilobytes (1024 bytes) consumed by the index file.

4.12 REMOVE(KEY.NO%,KEY.VALUE$,DATA.RECORD%,RETURN.CODE%)

REMOVE deletes KEY.VALUE$ from the index file specified by the KEY.NO%. The data record number associated with KEY.VALUE$ in the index is checked against DATA.RECORD% before the deletion takes place. If DATA.RECORD% is set to zero (0) before the call to REMOVE, then no such check takes place; and DATA.RECORD% is set to the associated record number corresponding to the KEY.VALUE$ to be deleted.

The RETURN.CODE% is set as follows:

0 if KEY.VALUE$ is not found in the index file;
1 for a successful deletion;
2 if DATA.RECORD% and the associated record number in the index do not agree in which case no deletion is performed; and
3 the key file specified by the KEY.NO% has not been ACCESSed.

If KEY.VALUE$ is a null string, then REMOVE takes no action; but RETURN.CODE% is set to one (1).

4.12.1 MICROSOFT BASIC EXAMPLE

```
10 INPUT "ENTER PART # TO BE DELETED:",PART.NO%
20 PART.NO$=MKI$(PART.NO%):REC.NO%=0
30 CALL REMOVE(PART.KEY%,PART.NO$,REC.NO%,RET.CODE%)
```

If the PART.NO$ is found, then it will be deleted from the index file corresponding to the value of PART.KEY%, and REC.NO% will be set to the corresponding data record number.

4.13 RSTRCT(KEY.NO%)

RSTRCT closes the index file corresponding to KEY.NO%. If any changes to the index file have occured since it was opened (ACCESSed), it is mandatory that RSTRCT be called to close the file. If not, the integrity of the index file is not ensured (since some updated nodes may still reside in an I/O buffer and/or the header record may be incorrect).

4.14 SEQUENTIAL LOADING OF INDEX

Three special MICRO B+™ routines are available to construct a B-Tree Index when existing data records can be accessed in sequential (key value) order. Data records can be accessed in

key value order if:

- they are physically stored in key value order; or
- there exist a set of pointers to the data records which provide logical access in key value order.

The computer time required to construct a B-Tree Index will be significantly reduced with these routines. But it MUST BE EMPHASIZED that these routines can be used only when BOTH of the following conditions are satisfied:

- there are no existing entries in the index; and
- the data records can be accessed in key value order.

Once an index has some entries, it is not possible to use these special routines unless the existing entries are merged with the new entries-to-be, and the existing index is ERAsed and replaced by a new, empty index file. Otherwise, routine ENTER must be used to add new entries to an existing index file; even if the new entries are in key sequential order.

4.14.1 **INTLOD**(KEY.NO%,LOAD.FACTOR%)

INTLOD initializes the MICRO B+ routines in preparation for the sequential building of a new B-Tree Index. INTLOD must be called after ACCESS and prior to the use of LOADKY (see next Section).

KEY.NO% specifies the index file to be constructed.

LOAD.FACTOR% specifies the number of entries to be loaded into each B-Tree node. LOAD.FACTOR% must be less than or equal to the maximum capacity of the nodes (see MAX.NO.KEYS% in Section 4.4), and greater than or equal to one-half the maximum capacity. As LOAD.FACTOR% increases, the size of the resultant index file decreases.

4.14.2 **LOADKY**(KEY.VALUE$,DATA.RECORD%,RETURN.CODE%)

Once INTLOD has been called to initialize the MICRO B+ routines (which presumes that INTREE and ACCESS have already been called), LOADKY is used to add the new KEY.VALUE$'s in increasing key value order.

KEY.VALUE$ is the string valued entity to be added to the index file specified by KEY.NO% in the call to INTLOD.

DATA.RECORD% is the pointer to the data file associated with the KEY.VALUE$.

RETURN.CODE% is set as follows:

- 1 for a successful load operation;
- 2 if KEY.VALUE$ already exists in the index, in which case the load operation is not performed; and
- 3 if KEY.VALUE$ is less than the previous key value added to the index, in which case the load operation is not performed.

4.14.2.1 IMPORTANT NOTE

Note that calls to LOADKY for different KEY.NO%'s are not permitted. The appropriate procedure is to CALL INTLOD for a specified key, followed by successive calls to LOADKY for the SAME key, followed by a call to BLDIND (see next Section).

4.14.3 BLDIND

BLDIND is called after all the sequentially orderd key values have been loaded via LOADKY. Note that LOADKY only loads the key values into the leaf-nodes of the B-Tree; BLDIND constructs the upper levels of the B-Tree. If BLDIND is not called, then a properly configured B-Tree will NOT exist. PLEASE NOTE that RSTRCT must still be called (after BLDIND) in order to properly close the index file.

4.14.4 MICROSOFT BASIC EXAMPLE

```
110 INPUT "# BUFFERS, # KEYS, # NODE SECTORS,
     # HEADER SECTORS:",NO.BUF%,NO.KEYS%,NO.SECT%,HD.SECT%
115 DAT.FILES%=0
120 CALL INTREE(NO.BUF%,NO.KEYS%,NO.SECT%,HD.SECT%,
     DAT.FILES%)
130 INPUT "KEY #,FILE NAME,LENGTH,TYPE,MAX CAPACITY:",
     KEY%,FILE.NAME$,KEY.LEN%,KEY.TYPE%,MAX.KV%
140 CALL ACCESS(KEY%,FILE.NAME$,KEY.LEN%,KEY.TYPE%,MAX.KV%)
150 INPUT "LOAD FACTOR:",LOAD.FAC%
160 CALL NMENTR(KEY%,CHECK.SIZE%)
170 IF CHECK.SIZE%<>0 THEN
     PRINT "ERROR...CHECK.SIZE%:";CHECK.SIZE%:STOP
180 CALL INTLOD(KEY%,LOAD.FAC%)
190 FOR I%=1 TO 10000
200    GET 1,I%
210    CALL LOADKY(KEY.VALUE$,I%,RET.CODE%)
220    IF RET.CODE%<>1 THEN
         PRINT "IMPROPER KEY VALUE:",KEY.VALUE$,I%,RET.CODE%
230 NEXT I%
240 CALL BLDIND
250 CALL RSTRCT(KEY%)
```

5.0 BASIC SOURCE CODE DATA FILE SUPPORT ROUTINES

While MICRO B+™ provides complete freedom of choice on how the data file is organized and managed, six MICROSOFT Basic data file support routines have been provided for the convenience of the application programmer. These routines open and close the data files, get space for new data records, report on the size of the data files, and return space from deleted records. The deleted data records are organized according to a stack (i.e., last in, first out) structure.

In addition to these routines, Section 6 describes similar routines implemented in assembly language which can be called from BASIC programs as well as from other languages.

To use the BASIC data file routines, follow these steps:

1. Set the global variable MICROB$ to the appropriate three (3) character code:

CODE	ROUTINE
ODF	OPEN DATA FILE
CDF	CLOSE DATA FILE
NWD	NEW DATA
OLD	RETURN DATA
DFS	DATA FILE SIZE
DFU	DATA FILE UTILIZATION

2. Set the appropriate input parameters to the desired values.

3. Execute a GOSUB 64000.

The input and output parameters, along with details of the routines are presented in the following Sections of Chapter 5.

5.1 OPEN DATA FILE

CODE	INPUT PARAMETERS	OUTPUT PARAMETERS
ODF	DATA.FILE$	NONE
	FILE.NO%	
	RECORD.LENGTH%	
	MESSAGE.LENGTH%	

This routine opens DATA.FILE$ with the specified RECORD.LENGTH% as FILE.NO%. If DATA.FILE$ does not exist, it will be created. The RECORD.LENGTH% must be the same each time the file is opened; and it must be at least 8 bytes. MESSAGE.LENGTH% should be set to the desired maximum length of any MESSAGE$ used in the RETURN DATA routine (see Section 5.4). MESSAGE.LENGTH% must be less than or equal to the smal-

ler of 255 or RECORD.LENGTH%-2. Note that the first record in DATA.FILE$ is a header record which maintains the status of the DATA.FILE$ stack structure. NO DATA CAN BE STORED IN THE FIRST RECORD OF DATA.FILE$!

Note that while FIELD statements are invoked by these routines, they only handle the management requirements of the data files. It is the programmer's responsibility to invoke FIELD statements to handle the actual data stored in the data files.

5.1.1 EXAMPLE

```
10 DATA.FILE$="D:CUSTOMER.DAT":FILE.NO%=4:MICROB$=
   "ODF":RECORD.LENGTH%=384:MESSAGE.LENGTH%=
   16:GOSUB 64000
```

This call would open the specified data file as file number 4 with a record length of 384 bytes. If no such file existed (on Drive D), a new file would be created.

5.2 CLOSE DATA FILE

CODE	INPUT PARAMETERS	OUTPUT PARAMETERS
CDF	FILE.NO%	NONE

This routine closes data files previously opened with a call to OPEN DATA FILE.

5.2.1 EXAMPLE

```
20 FILE.NO%=4:MICROB$="CDF":GOSUB 64000
```

5.3 NEW DATA

CODE	INPUT PARAMETERS	OUTPUT PARAMETERS
NWD	FILE.NO%	DATA.RECORD%

This routine returns the DATA.RECORD% number of the next available record in the DATA.FILE$ opened as FILE.NO%. The next available record is "popped" off the top of the stack for FILE.NO%. If the stack is empty, NEW DATA automatically increments the size of the data file to generate space for a new record. WHEN A RECORD IS POPPED OFF THE STACK, THE FIELDED VARIABLES FOR THE DATA FILE ARE CLOBBERED. THEREFORE, YOU MUST ASSIGN VALUES TO YOUR FIELDED VARIABLES FOR THE NEW DATA RECORD ONLY AFTER EXECUTING THE "NEW DATA" ROUTINE.

5.3.1 EXAMPLE

```
10 DATA.FILE$="D:CUSTOMER.DAT":FILE.NO%=4:MICROB$="ODF":
   RECORD.LENGTH%=128:MESSAGE.LENGTH%=10:
   GOSUB 64000
20 FIELD 4,2 AS CUST.NO$,18 AS LAST.NAME$,12 AS
   FIRST.NAME$,8 AS ACCT.RECV$
30 INPUT "ENTER CUSTOMER #,LAST NAME,FIRST NAME:",
   CUST.NO%,L.NAME$,F.NAME$
40 INPUT "ENTER ACCOUNTS RECEIVABLE:",AR#
50 FILE.NO%=4:MICROB$="NWD":GOSUB 64000
60 KEY$=MKI$(CUST.NO%):CALL ENTER(CUST.KEY%,KEY$,
   DATA.RECORD%,RET.CODE%)
70 KEY$=LEFT$(L.NAME$+SPACE$(9),9)+MKI$(CUST.NO%):
   CALL ENTER(NAME.KEY%,KEY$,DATA.RECORD%,CODE%)
80 LSET CUST.NO$=MKI$(CUST.NO%):LSET LAST.NAME$=L.NAME$:
   LSET FIRST.NAME$=F.NAME$:LSET ACCT.RECV$=MKD$(AR#)
90 PUT 4,DATA.RECORD%
```

DATA.RECORD% is set to the value of an empty record in the customer data file after executing statement 50. Then a new customer number and last name are added to the corresponding B-Trees with the associated record number given by DATA.RECORD% in statements 60 and 70. Note how the last name key was truncated and then made unique by the addition of the CUST.NO%. Finally, DATA.RECORD% in the customer data file is initialized by statements 80 and 90.

5.4 RETURN DATA

CODE	INPUT PARAMETERS	OUTPUT PARAMETERS
OLD	FILE.NO% DATA.RECORD% MESSAGE$	NONE

This routine "pushes" the returned DATA.RECORD% on to the top of the stack for FILE.NO%. After a call to RETURN DATA, the record in the data file with record number DATA.RECORD% has two fields written over the previous contents:

FIELD	LENGTH	DATA
1	2	LINK TO NEXT AVAILABLE RECORD
2	MESSAGE.LENGTH%	MESSAGE$

The MESSAGE$ parameter can be used to flag deleted (i.e., returned) records, or to save information from the deleted record for subsequent processing.

5.4.1 EXAMPLE

```
10 KEY$=MKI$(10231):DATA.RECORD%=0
15 CALL REMOVE(CUST.KEY%,KEY$,DATA.RECORD%,RET.CODE%)
20 IF DATA.RECORD%>0 THEN
     FILE.NO%=4:MESSAGE$="DELETED":MICROB$="OLD":
     GOSUB 64000
```

In this example, the data record corresponding to customer number 10231 would be returned to the data file for future use, and the customer number would be deleted from the B-Tree index; unless, of course, no such customer number were found.

5.5 DATA FILE SIZE

CODE	INPUT PARAMETERS	OUTPUT PARAMETERS
DFS	FILE.NO%	DATA.FILE.SIZE%

This routine returns the total number of records used by a data file including the header record and any returned (but unused) records.

5.5.1 EXAMPLE

```
10 FILE.NO%=4:MICROB$="DFS":GOSUB 64000
20 PRINT "THERE ARE ";DATA.FILE.SIZE%;" RECORDS ";
     "IN THE DATA FILE."
```

5.6 DATA FILE UTILIZATION

CODE	INPUT PARAMETERS	OUTPUT PARAMETERS
DFU	FILE.NO%	DATA.FILE.UTILIZATION%

This routine returns the number of records in a data file currently being used to store data. It excludes the header record and any returned (but unused) records.

5.7 MICRO B+™ DISK

These six data file management routines will be found in file

MDATA.BAS

on your MICRO B+™ disk. To include these routines in your

6.0 ASSEMBLY LANGUAGE DATA FILE SUPPORT ROUTINES

Nine assembly language data file support routines have been provided for the convenience of the application programmers. These routines may prove especially useful when working outside of a BASIC environment. These routines open and close the data files, get space for new data records, report on the size of the data files, return space from deleted records, and permit data file reads and writes. As with the BASIC source code routines described in the previous Section, it is not necessary to use these data file support routines with MICRO B+™. However, these routines do provide a simple method to organize the data files.

These assembly language routines restrict the record length to be multiples of 128 bytes.

Please note that the MICROSOFT Basic parameter conventions (as described in Section 4.1) apply to these routines.

Examples of the use of the following routines can be found in Section 9.

6.1 OPEND(FILE.NO%,FILE.NAME$,RECORD.SECTORS%)

OPEND opens or creates the data file given by FILE.NAME$ with a record length determined by the expression

RECORD.SECTORS% * 128.

The maximum value for RECORD.SECTORS% is 32 which means that the data files may have record lengths between 128 and 4096.

FILE.NO% is an integer between zero (0) and NO.DATA.FILES%-1, where NO.DATA.FILES% is the fifth parameter of INTREE (see Section 4.2). The maximum value for NO.DATA.FILES% is twenty (20). The remaining data file routines reference the data file using the FILE.NO% parameter. An attempt to assign the same FILE.NO% to more than one data file (at the same time) will result in a User Error.

The first record in each data file is reserved for a header record which maintains status information concerning the file. No data can be stored in the header record.

6.1.1 OPENR(FILE.NO%,FILE.NAME$,RECORD.SECTORS%)

Each time OPEND opens a data file, the integrity of the file is checked. A data file is considered to have lost its integrity if it is updated and not subsequently closed by CLOSED. If OPEND finds a loss of integrity, it will not open the data file; instead, a User Error occurs signaling the problem condition.

In order to facilitate the rebuilding of the data file, the routine OPENR is available. OPENR behaves exactly as OPEND except that no check of data file integrity is made. Hence, OPENR can be used to open a compromised data file for purposes of reconstruction.

6.2 **CLOSED**(FILE.NO%)

CLOSED closes the data file associated with FILE.NO%. If any updates have been made to the data file, it is necessary to call CLOSED. Otherwise, a User Error will result when the file is OPEND the next time.

6.3 **NEWDAT**(FILE.NO%,DATA.RECORD%)

NEWDAT returns in DATA.RECORD% the next available relative record number for the data file specified by FILE.NO%. If any previously deleted (see RETDAT) data records are availble, these are used first. If not, then NEWDAT increments the size of the data file to generate a new data record.

6.4 **RETDAT**(FILE.NO%,DATA.RECORD%)

RETDAT "pushes" the returned DATA.RECORD% onto the top of a logical stack of deleted data records. This stack is implemented via a linked list, which ensures fast operation since only the top most record is manipulated by the NEWDAT and RETDAT routines. After a call to RETDAT, the record in the data file with record number DATA.RECORD% has two fields written over the previous contents:

FIELD	LENGTH	DATA
1	1	FF Hex (255 decimal). This byte serves as flag for deleted data records as long as the user reserves this byte or ensures that no actual data will cause a FFH to occur in the first byte.
2	2	Link to next availbale data record.

6.5 **DATAFS**(FILE.NO%,FILE.SIZE%)

DATAFS returns in FILE.SIZE% the total number of data records in use or available for use in the file specified by FILE.NO%. The actual number of kilobytes consumed by the file can be

computed as

```
KILOBYTES% = ((FILE.SIZE% * RECORD.SECTORS%) + 7) / 8
```

6.6 **DATAFU**(FILE.NO%,UTILIZATION%)

DATAFU returns in UTILIZATION% the number of data records actually in use. The difference between the results returned by DATAFS and DATAFU represents the data records returned for re-use and the header record.

6.7 **READD**(FILE.NO%,DATA.RECORD%,BUFFER.PTR%)

READD reads the data record specified by the relative record number DATA.RECORD% into memory starting at the address contained in BUFFER.PTR%. It is up to the application program to ensure that sufficient space exists starting at the address given by BUFFER.PTR% to accept the data record. Also, it is up to the application program to be able to "parse" the data record into the desired variables for use by the application program. In languages that permit based variables and/or overlayed variables, this can be easily accomplished.

6.8 **WRITED**(FILE.NO%,DATA.RECORD%,BUFFER.PTR%)

WRITED writes the data record specified by DATA.RECORD% into the file specified by FILE.NO%, taking the information to write from memory starting at the address contained in BUFFER.PTR%.

Please note that the I/O does not occur at BUFFER.PTR%, but rather at the address contained in BUFFER.PTR%. For example, if one desired the information to be transferred from memory location 6000 hex, then

```
BUFFER.PTR% = 24576 (decimal)
```

7.0 HOW TO INTEGRATE MICRO B+™ WITH YOUR APPLICATION

MICRO B+™ is distributed as a library of relocatable object modules for 8080 or Z80 microprocessors operating under CP/M. The main requirement for using MICRO B+™ is a "linking loader" which accepts MICROSOFT formatted object module libraries. See MICROSOFT's "UTILITY SOFTWARE MANUAL" (Document No. 8401-334-01) for a complete description of their linking loader LINK-80 and the format of their relocatable object modules. In addition to a linking loader, it is helpful to have a MACRO assembler (such as MICROSOFT's MACRO-80 described in the above mentioned manual) which will allow you:

1) to create relocatable interface routines if your host language does not use the MICROSOFT parameter protocol and/or the MICROSOFT Basic string-descriptor vector (see Section 4.1); and

2) to easily change the size of the buffer area for index file I/O.

7.1 WHAT YOU GET ON YOUR MICRO B+™ DISK

Your MICRO B+™ disk contains the following files:

1) MICROB.REL -
   a library of relocatable object modules which contain the MICRO B+™ routines you call (e.g., ENTER) plus all the support routines required.

2) BUFMOD.REL -
   a relocatable object module which contains a prespecified buffer area of 3204 (decimal) bytes.

3) BUFMOD.MAC -
   an assembly language source module which contains the EQUate statement necessary to redefine the size of the buffer area.

4) MDATA.BAS -
   a series of MICROSOFT Basic source code routines to handle data file I/O.

Together, MICROB.REL and BUFMOD.REL contain everything you need to use the MICRO B+™ routines as described in Sections 4 and 6, provided you have an appropriate linking loader, and the 3204 byte buffer area is sufficient and/or not too large.

If you wish to create interface routines between your host language and the MICRO B+™ routines, an assembler which supports external declarations and entry point designations is required. Depending on the characteristics of your linking loader, it may be possible to customize the size of your buffer area (without setting up BUFMOD.MAC) by simply reser-

ving sufficient space at the end of the MICRO B+™ routines. In any event, it is very important to note that the MICRO B+™ routines expect the buffer area to begin at the entry point "FCB" which is supplied by BUFMOD.MAC.

7.1.1 MEMORY REQUIREMENTS OF MICRO B+™ CODE

The modular design of MICRO B+™ ensures that an application program utilizes only those portions of MICRO B+™ which are actually required. The "calls" to MICRO B+™ embedded in the application program determine which modules will be brought into the final .COM file by the linking loader. The memory requirements for the various modules comprising MICRO B+™ are given below:

MODULE COMPONENTS	MEMORY REQUIRED
INTREE,ACCESS,RSTRCT,RTRIEV,SEARCH SUCESR,PRDESR,NMENTR,NMNODE,SETERR	4.0K
ENTER	1.5K
REMOVE	1.7K
INTLOD,LOADKY,BLDIND	1.5K
OPEND,OPENR,CLOSED,NEWDAT,RETDAT DATAFS,DATAFU,READD,WRITED	1.0K

7.2 HOW LARGE A BUFFER AREA

The two primary factors affecting the buffer area size are the NO.NODE.SECTORS% and NO.BUFFERS%. Usually, NO.NODE.SECTORS% is set in accordance with the physical characteristics of the disk drives. The table below shows the most common choices for NO.NODE.SECTORS%.

PHYSICAL SECTOR SIZE OF DISK	COMMON CHOICES FOR NO.NODE.SECTORS%
128	2
256	2,4
512	2,4
1024	2,4,8

PLEASE NOTE THAT NO.NODE.SECTORS% SPECIFIES THE NUMBER OF 128 BYTE SECTORS COMPRISING EACH INDEX FILE RECORD. FOR EXAMPLE, A VALUE OF TWO (2) IMPLIES A 256-BYTE INDEX FILE RECORD LENGTH, REGARDLESS OF THE PHYSICAL SECTORING OF THE DISK DRIVE.

Within the choices given above, the selection is usually based on the key length. "Long" keys lead to higher values of

NO.NODE.SECTORS% in order to reduce the levels of the B-Tree.

For any specified NO.NODE.SECTORS%, the more buffers (i.e., larger NO.BUFFERS%), the fewer disk accesses required to retrieve a key value. However, when processing one index file at a time, the pay-off from adding buffers diminishes rapidly once five (5) or six (6) buffers are already in use. If more than one index file is in active use at one time, it is rewarding to increase the number of buffers even beyond six (6). Provided the memory space is available, figuring three (3) buffers per index file (in active use at the same time) is a reasonable rule-of-thumb. Please note that buffers are not assigned to individual index files. They are shared according to a least-recently-used priority which ensures that the active index files make full use of the buffers.

The final determinant of the size of the buffer area is the amount of memory available. If there is very little memory available for buffers, one can always reduce the NO.BUFFERS% down to the minimum level of three (3).

It is important to separate the concept of the size of the buffer area from the manner in which the buffer area is used. For any given specification of the maximum number of key files (NO.KEYS%), the record length of the key files (NO.NODE.SECTORS% * 128), the number of buffers (NO.BUFFERS%), and the number of data files suported by the assembly language routines of Section 6 (NO.DATA.FILES%), the required buffer size is given by

```
((NO.KEYS% + NO.DATA.FILES%) * 36) +
     (NO.BUFFERS% * ((NO.NODE.SECTORS% * 128) + 50))
```

For example, NO.KEYS%=6, NO.BUFFERS%=7, NO.DATA.FILES%=2, and NO.NODE.SECTORS%=4 require a buffer area of 4222 (decimal) bytes. Once this buffer space has been reserved, however, any combination of the four determining parameters which stays within 4222 bytes can be passed to the INTREE routine which sets up the way in which the buffer area is used.

Please note, however, that all key files used at the same time, i.e., after a call to INTREE and before any other subsequent calls to INTREE, must have identical record lengths.

7.3 STEP-BY-STEP SYSTEM INTEGRATION PLAN FOR COMPILED APPLICATIONS

7.3.1 CODE INTERFACE ROUTINES

If necessary, code interface routines in 8080 or Z80 assembly language. The basic form of the interface rou-

tines should be

- Reassign parameters according to the standard presented in Section 4.1.
- Call desired MICRO B+™ routine.
- Execute an 8080 RET.

If it is necessary to use interface routines to handle the parameter protocol of the host language, be sure to choose names for these routines that will not conflict with the MICRO B+™ routines.

Once coded, test interface routines to ensure that they pass the parameter addresses correctly.

7.3.2 PREPARE APPLICATION PROGRAM

Using the specifications from Sections 4,5 and 6 to guide you, code your application program including calls to the MICRO B+™ routines or your interface routines.

Be sure that your application program has:

1) a call to INTREE prior to any use of the index files;

2) initialized the string variables which will be set equal to index entries in SEARCH, SUCESR, and PRDESR;

3) ensured that all numeric KEY.VALUE$'s are of the specified KEY.LENGTH% (note that alphanumeric keys can be any length although they will be truncated or filled as necessary to achieve the KEY.LENGTH% whereas numeric keys must be of the exact KEY.LENGTH% before being used in any of the MICRO B+™ routines);

4) closed (via RSTRCT and/or CLOSED) all index and/or data files prior to subsequent calls to INTREE and/or terminating the application program.

7.3.3 CREATING BUFFER AREA

The single most important characteristic of the buffer area is that it must start at an entry point named "FCB". It is also important to note that the initialization routine INTREE does not check to see if the requested buffer area (as determined by NO.KEYS%, NO.BUFFERS%, NO.NODE.SECTORS%, and NO.DATA.FILES%) will fit within the

buffer area actually provided by BUFMOD.REL.

To redefine the buffer area to suit your application, use a text editor to change the operand of the EQUate statement in BUFMOD.MAC from

```
BUFSIZ   EQU    3204
```

to

```
BUFSIZ   EQU    WXYZ
```

where WXYZ is the decimal number of bytes required for the buffer area as determined by the formula in Section 7.2.

After redefining the BUFSIZ parameter, assemble BUFMOD.MAC to create a new relocatable module BUFMOD.REL. Please note that it is not necessary to name the buffer area relocatable module "BUFMOD.REL".

7.3.4 CREATING A COMPOSITE PROGRAM

Once you have compiled or assembled your application program (and interface routines, if needed), you can create a composite program which combines your application, the required modules from MICROB.REL, and the space for the index file buffer area.

Consider the following example:

1) You have compiled an application program resulting in a relocatable module called INVENTRY.REL.

2) You have assembled interface routines resulting in a module called INTRFACE.REL.

3) You have assembled, after appropriate redefinition, BUFMOD.MAC creating a buffer module called BUFMOD.REL.

Then the following MICROSOFT LINK-80 command sequence will create an executable program module called INVENTRY.COM:

```
L80 INVENTRY,INTRFACE,MICROB/S,BUFMOD,INVENTRY/N/E
```

where the LINK-80 program is assumed to reside in the program module L80.COM and all of the "REL" files have been assumed to reside on the logged-in disk. Further, it has been assumed that INVENTRY.REL does not require any other libraries (besides MICROB.REL) to be searched. Please see MICROSOFT's "UTILITY SOFTWARE MODULE" for a complete description of how to use their linking loader.

7.4 USING MICRO B+™ WITH THE MICROSOFT BASIC INTERPRETER

Provided one has a satisfactory linking loader, MICRO B+™ can be used with the MICROSOFT Basic Interpreter. (The linking loader supplied with the MICROSOFT Basic Compiler works well in this regard.)

The linking loader is used to place the MICRO B+™ routines in upper memory, and to determine the addresses of the user callable routines. Once the routines have been loaded, the interpreter can be invoked with an option which will protect the MICRO B+™ code.

To simplify this set-up procedure, several special relocatable modules are included on your MICRO B+™ disk:

INTRFACE.REL	NOSEQKEY.REL	INTRFCB.REL
NOADDKEY.REL	NODELKEY.REL	NODATA.REL

7.4.1 WHERE TO LOAD MICRO B+™

Once MBASIC (the interpreter) is invoked, you can determine the highest available memory location for MICRO B+™ by executing the statement

```
PRINT PEEK(7),PEEK(6)
```

which will return the address of the bottom of the your CP/M®. (Actually, this is the bottom of the permanent portion of CP/M®; it does not account for the CCP (Command Control Processor) which can, and will, be overwritten. For example, you might get

```
209      0
```

which is interpreted as D100 (hex). Note that D1 (hex) is equivalent to 209 (decimal).

Noting that the routines require about 4K bytes if only the retrieval and search routines are used, and about 7.5K bytes if all the routines (except those for sequential loading and data file I/O) are used, one can estimate the approximate origin for the routines. For example, one might want to use all the routines (with the above noted exception), plus a 2K byte buffer area. Then about 9.5K bytes would be required. Assuming the bottom of CP/M is D100, then the origin for MICRO B+™ would be AB00. (D100 - 2600 = AB00)

7.4.2. USING SPECIAL INTERFACE ROUTINES

Instead of simply loading the entire MICROB.REL library into memory, INTRFACE.REL provides a mechanism to allow only those routines actually needed to be loaded. Further, it simplifies determining the addresses of the routines which are used in the CALL statements.

For example, to load the routines at AB00 as suggested above, the following command would be used with MICROSOFT'S Loader:

```
L80 /P:AB00,INTRFACE,NOSEQKEY,NODAT,MICROB/S,INTRFCB/E
```

This will load all the routines (except those for sequential loading of key values and data file I/O), set the origin of the buffer area, and exit back to the monitor.

INTRFACE.REL contains jumps to all the user callable routines in MICROB.REL. INTRFCB.REL contains the origin for the buffer area. NOSEQKEY.REL contains a dummy set of routines for sequential loading so that the loader does not select these routines when searching MICROB.REL. Similarly, NOADDKEY.REL will keep the routines to add a new key value from being loaded, NODELKEY.REL will keep the routines to remove a key value from being loaded, and NODAT.REL will keep the data file routines of Section 6 from being loaded. For example, to load only the search and retrieval routines, one would use

```
L80 /P:AB00,INTRFACE,NOSEQKEY,NOADDKEY,NODELKEY,
     NODAT,MICROB/S,INTRFCB/E
```

7.4.3 PROTECTING THE CODE FROM MBASIC

To protect the routines loaded as above, the "/M" switch is used when MBASIC is invoked. Continuing with our example in which the routines are loaded at AB00, the interpreter should be invoked as follows:

```
MBASIC /M:&HAAFF
```

which causes location AAFF (hex) to be the last byte used by the interpreter.

Finally, it is necessary to assign the routine names (e.g., ACCESS%) with the proper addresses. If INTRFACE.REL is used, then the following assignment statements can be used at the beginning of your application program (NOTE THAT THE ORIGIN USED HERE [AB00] IS BASED ON OUR EXAMPLE):

```
10 ORG%=&HAB00:INTREE%=ORG%:ACCESS%=ORG%+3:ACCES1%=ORG%+6
20 ACCES2%=ORG%+9:ENTER%=ORG%+12:RTRIEV%=ORG%+15:SEARCH%=ORG%+18
30 SUCESR%=ORG%+21:NMENTR%=ORG%+24:REMOVE%=ORG%+27:RSTRCT%=ORG%+30
40 INTLOD%=ORG%+33:LOADKY%=ORG%+36:BLDIND%=ORG%+39:NMNODE%=ORG%+42
50 SETERR%=ORG%+45:PRDESR%=ORG%+48:OPEND%=ORG%+51:OPENR%=ORG%+54
60 CLOSED%=ORG%+57:NEWDAT%=ORG%+60:RETDAT%=ORG%+63:DATAFS%=ORG%+66
70 DATAFU%=ORG%+69:READD%=ORG%+72:WRITED%=ORG%+75
```

Please note in the above assignments that the addresses of the various MICRO B+™ routines are assigned to INTEGER variables. Therefore, be sure to use a statement of the form

```
100 CALL RTRIEV%(KEY.NO%,TARGET$,DATA.RECORD%)
```

in which the name of the routine is actually an integer variable. When moving this code to the compiler, it will be necessary to strip the percent ("%") signs from the routine names as well as eliminating the assignment statements in lines 10 through 70 above.

7.4.4 PROTECTING THE CODE FROM THE CCP

Loading the index file routines into upper memory for subsequent use with the interpreter causes the CCP (Command Control Processor) to be overwritten. After the load operation is completed, the CCP is written back into memory (in order to process your next command) which causes the end of the index file routines to be overwritten! However, if the buffer area is at least 2K bytes, the CCP will not disturb any of the MICRO B+™ code. Therefore, be sure to:

- use a buffer at least as large as the CCP (usually 2K); and
- place the buffer area at the end of the code.

7.4.5 SAVE YOUR PROGRAM

It is very important to note that you should save your program before attempting to run it if you are making CALLs to MICRO B+™. Unlike errors detected by the interpreter, User Errors return you to CP/M (unless you have called SETERR), and you will not be able to save your code. Therefore, be sure to save your code prior to running it.

8.0 ERROR CODES

The MICRO B+™ routines generate two types of error codes:

- MICRO B+™ Error Codes; and

- User Error Codes.

MICRO B+™ Error Codes are generated when a B-Tree does not satisfy the internal consistency checks which the MICRO B+™ routines perform whenever an index file is used. Such errors should never occur. If a MICRO B+™ Error does occur, an error message of the form

```
MICRO B+™ Error ...XY...
```

will be displayed. If, after careful review of your application program, you cannot find any obvious cause for the MICRO B+™ error (such as using an index file which was not properly closed by RSTRCT, or trying to build an index file greater than 512K bytes under CP/M® 1.4), please contact FairCom. You should include as much documentation as possible, including the two character error code which is displayed.

8.1 USER ERROR CODES

User Errors occur when avoidable problems occur (such as no more room on a disk or an illegal KEY.NO%). User Errors can be handled in two ways:

1) If SETERR (see Section 4.3) is called near the beginning of an application program, User Errors can be trapped by testing the ERROR.CODE% variable (passed to SETERR) for a non-zero value after calls to MICRO B+™ routines.

2) If SETERR is not called, or prior to calling SETERR, or after calling SETERR with a zero parameter, a User Error results in a message the form

```
USER ERROR ...XY... CHECK MANUAL
```

being sent to the console device. Further, a warm boot is executed which causes the application program to lose control.

An explanation of the ERROR.CODE% values and the corresponding two-character User Error Codes follows:

VALUE	CODE	EXPLANATION
20	AD	Could not write an index file record during CP/M 1.4 processing. Possible causes are: disk full or directory full.
21	AE	Could not write an index file record during CP/M 2.x processing. See #20.
22	AF	Could not read index file record during CP/M 1.4 processing. Possible cause is attempting to use a newly created index file which has not been properly closed.
23	AG	Could not read index file record during CP/M 2.x processing. See #22.
24	AH	No more directory space. Occurred while trying to create new index file.
30	AN	KEY.NO% out of range. KEY.NO% must satisfy 0 <= KEY.NO% < NO.KEYS%
31	AO	Illegal INDEX.FILE$ parameter in call to ACCESS. Most likely causes are a null index file name, or an exceptionally long index file name.
33	BA	Attempt to reuse a KEY.NO% already assigned to an ACCESSed index file.
34	BB	KEY.LENGTH% parameter in ACCESS exceeds maximum allowable value of 48.
35	BC	INDEX.KEY$ parameter used in SEARCH, SUCESR, and PRDESR has not been properly initialized. INDEX.KEY$ must be at least as "long" as the key length.
36	BD	LOAD.FACTOR% out of range. LOAD.FACTOR% must satisfy MAX.KV%/2 <= LOAD.FACTOR% < MAX.KV%
37	BE	INTLOD called before ACCESS.
38	BF	BLDIND not properly called.

VALUE	CODE	EXPLANATION
39	BG	MAX.KV% too large for KEY.LENGTH% and NO.NODE.SECTORS%.
40	BH	Corrupted index file detected during call to ACCESS. Index file becomes corrupted if it is updated (i.e., additions or deletions have occurred), and not subsequently closed via RSTRCT. Index file must be rebuilt.
50	CB	Could not write data file record during CP/M 1.4 processing. Possible causes are: disk full or directory full.
51	CC	Could not write data file record during CP/M 2.x processing. See #50.
52	CD	Could not read data file record during CP/M 1.4 processing. Possible causes are: an attempt to use a newly created data file which has not been properly CLOSED, or a read past end-of-file.
53	CE	Could not read data file record during CP/M 2.x processing. See #52.
54	CF	No more directory space. Occurred while trying to create new data file.
55	CG	Could not close data file. Occurred during call to CLOSED.
60	CL	FILE.NO% out of range. FILE.NO% must satisfy: 0 <= FILE.NO% < NO.DATA.FILES%
61	CM	Illegal FILE.NAME$ parameter in call to OPEND or OPENR. See #31.
63	CO	Attempt to reuse a FILE.NO% assigned to a data file in use.
65	DA	Too many RECORD.SECTORS% in call to OPEND or OPENR. RECORD.SECTORS% must be no greater than 32.

9.0 RECREATE & DATABASE: TWO USEFUL EXAMPLES

The two example programs presented here are useful in their own right as well as demonstrating the use of MICRO B+™. RECREATE.BAS can be used to easily recreate a data file and its associated index files if either the data file or its indices become corrupted. DATABASE.BAS allows one to build a name and address database. The database entries may be examined, updated, and/or listed. The program itself can be expanded to meet your particular requirements.

9.1 RECREATE.BAS

RECREATE.BAS is designed to permit easy reconstruction of a data file and its associated indices. As presented, it assumes that the data file and the indices must be recreated. It can be changed so that only the indices are rebuilt.

Both RECREATE.BAS and RECREATE.COM will be found on your MICRO B+™ disk. RECREATE.COM can be used to rebuild files even if they were created by application programs written in a language other than MICROSOFT BASIC, provided that the data files are compatible with the routines described in Section 6. Of course, RECREATE.BAS can be adapted to almost any programming environment, and any data file organization.

The version of RECREATE.BAS on your disk may be a latter version than that presented below. You should consider the version on your disk to be the definitive version.

Please note that the underlines ("_") are continuation marks; they permit a logical program line to be spread over multiple physical lines.

```
 900 REM ***** ************************************************************
 901 REM
 902 REM         RECREATE UTILITY    VERSION 2.1     2/19/81
 903 REM
 904 REM ************************************************************************
1000 DIM INDEX.NAME$(9),KEY.LENGTH%(9),KEY.TYPE%(9),MAX.KV%(9)
1010 DIM REC.SUFFIX$(9),KEY.POSITION%(9),KEY.VALUE$(9)

1100 INPUT "Enter OLD Data File Name:",CUR.DAT.FILE$
1110 INPUT "Enter NEW Data File Name:",REP.DAT.FILE$
1120 INPUT _
      "Enter the number of 128 byte sectors per data file record:", _
        REC.SECTORS%
1130 INPUT _
      "Enter beginning data record number (e.g.,1 or 2):", _
        BEG.REC% 'You can skip header records
```

```
1140 INPUT _
      "Do you want to eliminate records beginning with ffH (Y/N):", _
        DEL.FF$ 'RETDAT automatically sets 1st byte to ffH
1150 IF DEL.FF$="y" THEN DEL.FF$="Y"
1160 NO.DAT.FILES%=2
1170 PRINT

1200 INPUT "Enter the number of Key Fields:",NO.KEYS%
1210 INPUT "Enter the number of Node Sectors & Header Sectors:", _
        NODE.SECTORS%,HEADER.SECTORS%
1220 INPUT "Enter the number of Index File Buffers:",NO.BUFFERS%
1230 REQ.BUFFER%=(NO.KEYS%+NO.DAT.FILES%)*36+ _
        NO.BUFFERS%*(NODE.SECTORS%*128+50)
1240 IF REQ.BUFFER% > 8192 THEN _
        PRINT :PRINT "NOT ENOUGH BUFFER SPACE": _
        GOTO 1160 _
     ELSE _
        PRINT :PRINT TAB(5);REQ.BUFFER%; _
          " bytes of buffer space utilized."
1260 CALL INTREE(NO.BUFFERS%,NO.KEYS%,NODE.SECTORS%,HEADER.SECTORS%, _
        NO.DAT.FILES%) 'Initialize MICRO B+(tm)
1270 ERROR.CODE%=1
1280 CALL SETERR(ERROR.CODE%) 'Enable error trapping

1300 CUR.DAT.FILE%=0 'Assign data file number
1310 REP.DAT.FILE%=1 'Assign data file number
1315 REM  Note how OPENR has been used to open a potentially cor-
1316 REM  rupted data file.
1320 CALL OPENR(CUR.DAT.FILE%,CUR.DAT.FILE$,REC.SECTORS%)
1330 IF ERROR.CODE%<>0 THEN ERROR.TYPE%=1:GOTO 9000
1340 CALL OPEND(REP.DAT.FILE%,REP.DAT.FILE$,REC.SECTORS%)
1350 IF ERROR.CODE%<>0 THEN ERROR.TYPE%=2:GOTO 9000
1360 PRINT
1370 PRINT _
      "Enter Index Name,Key Length,Type(0/1),MaxKV/Node,Auto Suffix"; _
        " (Y/N),Key Position"
1380 PRINT
1390 FOR KEY.NO%=0 TO NO.KEYS%-1 'Input index file parameters
1400 REM  Auto Suffix will add a two-byte representation of the data
1401 REM  record number to the end of a key (to ensure uniqueness).
1402 REM
1403 REM  Key Position specifies the starting column of the key field.
1404 REM  For example, if a key occupies the 5th through 10th bytes of
1405 REM  the data record, enter a "5" for Key Position.
1410 PRINT TAB(6);"For Key";KEY.NO%;">>";
1420    INPUT " ",INDEX.NAME$(KEY.NO%),KEY.LENGTH%(KEY.NO%),KEY.TYPE%(_
              KEY.NO%),MAX.KV%(KEY.NO%),REC.SUFFIX$(KEY.NO%), _
              KEY.POSITION%(KEY.NO%)
1425    IF REC.SUFFIX$(KEY.NO%)="y" THEN REC.SUFFIX$(KEY.NO%)="Y"
1430    CALL ACCESS(KEY.NO%,INDEX.NAME$(KEY.NO%),KEY.LENGTH%(_
              KEY.NO%),KEY.TYPE%(KEY.NO%),MAX.KV%(KEY.NO%))
1440    IF ERROR.CODE% <> 0 THEN ERROR.TYPE%=3:GOTO 9000
1450 NEXT KEY.NO%
```

```
1500 OPEN "R",1,"DUMMY.$$$",REC.SECTORS%*128
1505 REM  A duumy file provides a buffer area and the ability to
1506 REM  use FIELDed variables.
1510 FIELD 1,1 AS DEL.CHK$
1520 FOR KEY.NO%=0 TO NO.KEYS%-1 'Set up key fields
1530    FIELD 1,KEY.POSITION%(KEY.NO%)-1 AS DUMMY$, _
          KEY.LENGTH%(KEY.NO%) AS KEY.VALUE$(KEY.NO%)
1540 NEXT KEY.NO%
1550 BUFFER.PTR%=VARPTR(#1) 'Determine position of data file buffer
1560 PRINT
1570 PRINT "=== Please wait while data is processed. ===": PRINT

2000 CUR.REC.NO%=BEG.REC%:REP.REC.NO%=0 'Set starting point in files
2010 CALL READD(CUR.DAT.FILE%,CUR.REC.NO%,BUFFER.PTR%)
2020 WHILE ERROR.CODE% = 0 'Loop until end-of-file on old data file
2030    IF DEL.CHK$=CHR$(255) AND _
          DEL.FF$="Y" THEN GOTO 2140 'Skip deleted records
2035     CALL NEWDAT(REP.DAT.FILE%,REP.REC.NO%)
2040    REP.REC.NO$=MKI$(REP.REC.NO%) 'For use with Auto Suffix
2050    FOR KEY.NO%=0 TO NO.KEYS%-1 'Add each key to index file
2051    REM
2052    REM  The calls to READD bring the data into the I/O buffer
2053    REM  pointed to by BUFFER.PTR%. This automatically causes
2054    REM  the FIELDed variables "KEY.VALUE$()" to be set to the
2055    REM  data fields described by KEY.POSITION% and KEY.LENGTH%.
2056    REM
2060            KEY$=KEY.VALUE$(KEY.NO%)
2070            IF REC.SUFFIX$(KEY.NO%)="Y" THEN _ 'Add suffix
                    KEY$=LEFT$(KEY$,KEY.LENGTH%(_
                          KEY.NO%)-2)+REP.REC.NO$
2080            CALL ENTER(KEY.NO%,KEY$,REP.REC.NO%,RET.CODE%)
2090            IF ERROR.CODE% <> 0 THEN ERROR.TYPE%=4:GOTO 9000
2100            IF RET.CODE% <> 1 THEN _ 'Should not occur
                    PRINT "RETURN CODE,KEY.NO%,OLD REC #,"; _
                    "NEW REC#:",RET.CODE%;KEY.NO%,CUR.REC.NO%,_
                    REP.REC.NO%
2110    NEXT KEY.NO%
2120    CALL WRITED(REP.DAT.FILE%,REP.REC.NO%,BUFFER.PTR%)
2130    IF ERROR.CODE% <> 0 THEN ERROR.TYPE%=5:GOTO 9000
2140    CUR.REC.NO%=CUR.REC.NO%+1
2150    CALL READD(CUR.DAT.FILE%,CUR.REC.NO%,BUFFER.PTR%)
2160 WEND
2170 FOR KEY.NO%=0 TO NO.KEYS%-1
2180    CALL RSTRCT(KEY.NO%)
2190 NEXT KEY.NO%
2200 CALL CLOSED(REP.DAT.FILE%)
2210 PRINT :PRINT "=== RECREATE TERMINATING ==="
2220 CUR.REC.NO%=CUR.REC.NO%-BEG.REC%
2230 PRINT TAB(10);"AFTER";CUR.REC.NO%;"RECORDS PROCESSED"
2240 PRINT TAB(10);"LAST RECORD IN NEW FILE IS #";REP.REC.NO%
2250 PRINT TAB(10);"TERMINATION CODE IS";ERROR.CODE%
2260 CLOSE 1
2270 KILL "DUMMY.$$$"
2280 STOP
```

```
9000 PRINT :PRINT "User Error #";ERROR.CODE%; _
     "occurred while trying to ";
9010 ON ERROR.TYPE% GOTO 9100,9200,9300,9400,9500
9100 PRINT "open file: ";CUR.DAT.FILE$ :STOP
9200 PRINT "open file: ";REP.DAT.FILE$ :STOP
9300 PRINT "ACCESS index: ";INDEX.NAME$(KEY.NO%) :STOP
9400 PRINT "add a key value to: ";INDEX.NAME$(KEY.NO%)
9410 PRINT "The key value came from record #";CUR.REC.NO%; _
     " in ";CUR.DAT.FILE$
9420 STOP
9500 PRINT "write a record to ";REP.DAT.FILE$
9510 PRINT "The data came from old record #";CUR.REC.NO%;" and was to go"
9520 PRINT "in new record #";REP.REC.NO%
9530 STOP
```

Some features of RECREATE.BAS worth noting are:

- the use of a dummy data file to allow use of a buffer area and FIELDed variables;
- the use of OPENR to allow a corrupted data file to be opened for rebuilding;
- the way in which the data record number can be used as a "tie-breaker" for non-unique keys;
- the use of "IF" statements to trap possible User Errors;

9.2 DATABASE.BAS

DATABASE.BAS allows you to build and maintain a data file nominally organized by last name, zipcode, and customer number. It has been designed so that other attributes can be easily made into "keys." The primary purpose for presenting DATABASE.BAS is to provide a rich environment for examples of how to use the MICRO B+™ routines.

The version of DATABASE.BAS on your disk may be a latter version than that presented below. You should consider the version on your disk to be the definitive version.

Again, the underlines ("_") represent continuation marks.

```
 900 REM ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
 901 REM
 902 REM        DATABASE EXAMPLE   VERSION 2.1  2/19/81
 903 REM
1000 REM ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
1010 REM
1020 REM         SET-UP DATABASE FIELD & KEY DESCRIPTORS
1030 REM
1040 REM ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
1050 DIM FLD.NAME$(7),FLD.LEN%(7),NEW.FLD$(7),OLD.FLD$(7)
1060 MAX.FIELD%=7:NO.FIELDS%=MAX.FIELD%+1
1070 FLD.NAME$(0)="Customer Number" :FLD.LEN%(0)=4
1080 FLD.NAME$(1)="First Name"      :FLD.LEN%(1)=16
1090 FLD.NAME$(2)="Last Name"       :FLD.LEN%(2)=20
1100 FLD.NAME$(3)="Street Address"  :FLD.LEN%(3)=20
1110 FLD.NAME$(4)="City"            :FLD.LEN%(4)=20
1120 FLD.NAME$(5)="State"           :FLD.LEN%(5)=2
1130 FLD.NAME$(6)="Zipcode"         :FLD.LEN%(6)=9
1140 FLD.NAME$(7)="Customer Status" :FLD.LEN%(7)=36

1200 DIM KEY.NAME$(2),KEY.LEN%(2),KEY.MAP%(2),KEY.TYPE%(2),MAX.KV%(2)
1210 MAX.KEY%=2 :NO.KEYS%=MAX.KEY%+1
1220 KEY.LEN%(0)=10:KEY.TYPE%(0)=0:KEY.MAP%(0)=2 ' KEY 0 = LAST NAME
1230 KEY.LEN%(1)=11:KEY.TYPE%(1)=0:KEY.MAP%(1)=6 ' KEY 1 = ZIPCODE
1240 KEY.LEN%(2)=2 :KEY.TYPE%(2)=1:KEY.MAP%(2)=0 ' KEY 2 = CUSTOMER NUMBER
1245 UNIQ.KEY%=2 'USED IN TEST OF UNIQUENESS
1250 FOR KEY%=0 TO MAX.KEY% 'SET KEY NAMES TO CORRESPONDING FIELD NAMES
1260    KEY.NAME$(KEY%)=FLD.NAME$(KEY.MAP%(KEY%))
1270 NEXT KEY%

1300 DIM INDEX.NAME$(2)
1310 INDEX.NAME$(0)="NAME.IDX"
1320 INDEX.NAME$(1)="ZIPC.IDX"
1330 INDEX.NAME$(2)="NUMB.IDX"

2000 REM
2010 REM ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
2020 REM
2030 REM                INITIALIZE INDEX FILES
2040 REM
2050 REM ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
2060 YES%=-1 :NO%=0
2070 INDEX.KEY$=SPACE$(11) 'SET TO LONGEST KEY LENGTH
2080 NO.BUFFERS%=6 :NO.NODE.SECTORS%=2 :NO.HEADER.SECTORS%=2
2090 NO.DATA.FILES%=1
2100 CALL INTREE(NO.BUFFERS%,NO.KEYS%,NO.NODE.SECTORS%, _
                 NO.HEADER.SECTORS%,NO.DATA.FILES%)
2110 ERROR.CODE%=1 :CALL SETERR(ERROR.CODE%) 'TRAP USER ERRORS
```

```
2120 FOR KEY%=0 TO MAX.KEY% 'OPEN INDEX FILES
2130    MAX.KV%(KEY%)=(NO.NODE.SECTORS%*128-8)/(KEY.LEN%(KEY%)+2)
2140    MAX.KV%(KEY%)=MAX.KV%(KEY%)/2*2 'MAKE SURE ITS EVEN
2150    CALL ACCESS(KEY%,INDEX.NAME$(KEY%),KEY.LEN%(KEY%), _
                    KEY.TYPE%(KEY%),MAX.KV%(KEY%))
2160    IF ERROR.CODE%<>0 THEN ERROR.TYPE%=1:GOTO 9000
2170 NEXT KEY%

3000 REM
3010 REM :::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
3020 REM
3030 REM                INITIALIZE DATA FILE
3040 REM
3050 REM :::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
3060 FILE.NO%=0 :RECORD.SECTORS%=1 ' 128 BYTE DATA FILE RECORD LENGTH
3070 FILE.NAME$="CUSTOMER.DAT"
3080 CALL OPEND(FILE.NO%,FILE.NAME$,RECORD.SECTORS%)
3090 IF ERROR.CODE%<>0 THEN ERROR.TYPE%=2:GOTO 9000
3100 REM        IN ORDER TO ESTABLISH A BUFFER AREA FOR THIS FILE,
3110 REM        WE WILL OPEN A 'DUMMY' MBASIC FILE.
3120 OPEN "R",1,"DUMMY.$$$",RECORD.SECTORS%*128
3130 BUFFER.PTR%=VARPTR(#1)
3140 FIELD #1,1 AS DEL.FLAG$,2 AS CUST.NO$,FLD.LEN%(1) AS F.NAME$, _
     FLD.LEN%(2) AS L.NAME$,FLD.LEN%(3) AS ADDRESS$,FLD.LEN%(4) AS CITY$
3150 DUMMY%=3+FLD.LEN%(1)+FLD.LEN%(2)+FLD.LEN%(3)+FLD.LEN%(4)
3160 FIELD #1,DUMMY% AS DUMMY$,FLD.LEN%(5) AS STATE$,FLD.LEN%(6) AS ZIPCODE$,_
     FLD.LEN%(7) AS STATUS$

4000 REM
4010 REM :::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
4020 REM
4030 REM                BEGIN DATABASE OPERATION
4040 REM
4050 REM :::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
4060 GOSUB 10000 'CLEAR SCREEN
4070 GOSUB 11000 'PRINT MAIN MENU & GET CHOICE
4080 ON CHOICE% GOTO 5100,5300,5500,5700,5900,6100

5100 REM
5102 REM :::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
5104 REM
5106 REM                ENTER NEW CUSTOMERS
5108 REM
5110 REM :::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
5115 ENTER.MODE$="NEW" :GOSUB 12000 'DATA ENTRY ROUTINE
5120 IF ACTION$="SAVE" THEN _
             DATA.RECORD%=0 : _ 'SIGNAL NEED FOR A NEW DATA RECORD
             GOSUB 13000    : _ 'UPDATE INDICES & DATA FILE
             GOTO 5115
5125 GOTO 4060 'RETURN TO MENU
```

```
5300 REM
5302 REM ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
5304 REM
5306 REM                SCAN/UPDATE/DELETE CUSTOMERS
5308 REM
5310 REM ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
5315 GOSUB 14000 'DETERMINE SEARCH KEY
5320 KEY%=CHOICE% :PRINT
5325 PRINT "Enter target value for ";KEY.NAME$(KEY%);","
5330 LINE INPUT "     or press 'RETURN' to see main menu>>",TARGET$
5335 IF TARGET$="" THEN 4060 'RETURN TO MAIN MENU
5340 GOSUB 15000 'CONVERT TARGET TO KEY FORMAT
5345 CALL SEARCH(KEY%,CONV.TARGET$,DRN%,INDEX.KEY$)
5350 IF ERROR.CODE%<>0 THEN ERROR.TYPE%=3:GOTO 9000
5355 CONTINUE%=YES%
5360 WHILE CONTINUE% AND DRN%<>0
5365    GOSUB 16000 'READ CUSTOMER RECORD # DRN%
5370    ENTER.MODE$="OLD" :GOSUB 12000 'DATA ENTRY ROUTINE
5372    DATA.RECORD%=DRN% : SAVE.KEY%=KEY%
5375    IF ACTION$="SAVE" THEN GOSUB 13000 'UPDATE INDICES & DATA FILE
5380    IF ACTION$="DELT" THEN GOSUB 17000 'DELETE ENTRY
5385    IF ACTION$="SAVE" OR ACTION$="DELT" THEN _ 'RESET INTERNAL PTR
            KEY%=SAVE.KEY% : _
            CONV.TARGET$=LEFT$(INDEX.KEY$,KEY.LEN%(KEY%)) : _
            CALL SEARCH(KEY%,CONV.TARGET$,DRN%,INDEX.KEY$) : _
            IF ERROR.CODE%<>0 THEN ERROR.TYPE%=13:GOTO 9000
5390    IF ACTION$="CONT" THEN CALL SUCESR(KEY%,DRN%,INDEX.KEY$)
5395    IF ACTION$="BACK" THEN CALL PRDESR(KEY%,DRN%,INDEX.KEY$)
5400    IF ACTION$="STOP" THEN CONTINUE%=NO%
5405 WEND
5410 PRINT
5415 PRINT "SCAN ENDED"
5420 GOSUB 18000 'PAUSE
5425 GOTO 4060 'RETURN TO MAIN MENU

5500 REM
5502 REM ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
5504 REM
5506 REM                        LIST CUSTOMERS
5508 REM
5510 REM ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
5515 GOSUB 14000 'DETERMINE SEARCH KEY
5520 KEY%=CHOICE%
5525 PRINT
5530 LINE INPUT "Do you want listing routed to printer (Y/N)>>",ROUTE$
5535 IF ROUTE$="y" THEN ROUTE$="Y" 'DEFAULT = NO
5540 PRINT
5545 PRINT "Enter lower and upper limits for ";KEY.NAME$(KEY%);" listing;"
5550 INPUT "     separate values with a comma >>", _
            L.VALUE$,U.VALUE$
5555 TARGET$=L.VALUE$:GOSUB 15000:L.VALUE$=CONV.TARGET$ 'CONVERT TO KEY FORMAT
5560 TARGET$=U.VALUE$:GOSUB 15000:U.VALUE$=CONV.TARGET$
5565 CALL SEARCH(KEY%,L.VALUE$,DRN%,INDEX.KEY$)
5570 NO.LISTED%=0
```

```
5575 IF DRN%<>0 THEN GOSUB 19000 'COMPARE INDEX.KEY&U.VALUE. RETURNS COMPARE%.
5580 WHILE DRN%<>0 AND COMPARE%<=0
5585    GOSUB 16000 'READ CUSTOMER RECORD # DRN%
5590    GOSUB 25000 'PRINT RECORD
5595    NO.LISTED%=NO.LISTED%+1
5600    CALL SUCESR(KEY%,DRN%,INDEX.KEY$)
5605    IF DRN%<>0 THEN GOSUB 19000 'COMPARE INDEX.KEY & U.VALUE
5610 WEND
5615 PRINT
5620 PRINT TAB(5);NO.LISTED%;" records listed."
5625 GOSUB 18000 'PAUSE
5630 GOTO 4060 'RETURN TO MAIN MENU

5700 REM
5702 REM ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
5704 REM
5706 REM                DATABASE STATISTICS
5708 REM
5710 REM ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
5715 CALL DATAFS(FILE.NO%,FILE.SIZE%) 'NOTE FILE.SIZE INCLUDES HEADER
5720 CALL DATAFU(FILE.NO%,FILE.UTIL%) 'FILE.UTIL DOES NOT
5725 GOSUB 10000 'CLEAR SCREEN
5730 PRINT TAB(5);FILE.NAME$;" has ";FILE.SIZE%;" records; currently, ";
5735 PRINT FILE.UTIL%;" of them are in use."
5740 PRINT :PRINT :PRINT :PRINT
5745 PRINT TAB(5);"INDEX";TAB(30);"ENTRIES"
5750 PRINT TAB(5);"----------------";TAB(30);"-------"
5755 FOR KEY%=0 TO MAX.KEY%
5760    CALL NMENTR(KEY%,NO.ENTRIES%)
5765    PRINT TAB(5);KEY.NAME$(KEY%);TAB(32);NO.ENTRIES%
5770 NEXT KEY%
5775 PRINT :PRINT :PRINT :PRINT
5780 GOSUB 18000 'PAUSE
5785 GOTO 4060 'RETURN TO MAIN MENU

5900 REM
5902 REM ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
5904 REM
5906 REM                SAVE DATABASE UPDATES & RESTART
5908 REM
5910 REM ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
5912 RESTART%=YES%
5915 CALL CLOSED(FILE.NO%)
5920 IF ERROR.CODE%<>0 THEN ERROR.TYPE%=4:GOTO 9000
5925 FOR KEY%=0 TO MAX.KEY%
5930    CALL RSTRCT(KEY%)
5935    IF ERROR.CODE%<>0 THEN ERROR.TYPE%=5:GOTO 9000
5940 NEXT KEY%
5945 CLOSE 1 'CLOSE DUMMY FILE
5950 KILL "DUMMY.$$$"
5955 IF RESTART% THEN GOTO 2000
6000 PRINT
6005 PRINT "*** SUCCESSFUL TERMINATION ***"
6010 STOP
```

```
6100 REM
6102 REM ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
6104 REM
6106 REM                  SAVE DATABASE UPDATES & TERMINATE
6108 REM
6110 REM ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
6115 RESTART%=NO%
6120 GOTO 5915

9000 REM
9010 REM :::::::::::::::: ::::::::::::::::::::::::::::::::::::::::::::::::::::
9020 REM
9030 REM                  ERROR HANDLING
9040 REM
9050 REM ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
9100 PRINT
9110 PRINT "User Error #";ERROR.CODE%;" occurred while trying to ";
9120 ON ERROR.TYPE% GOTO 9210,9220,9230,9240,9250,9260,9270,9280,9290, _
       9300,9310,9320,9330
9210 PRINT "access ";INDEX.NAME$(KEY%) : GOTO 9700
9220 PRINT "open ";FILE.NAME$ : GOTO 9700
9230 PRINT "search ";KEY.NAME$(KEY%);" Index File" : GOTO 9500 'CLOSE FILES
9240 PRINT "close ";FILE.NAME$ : GOTO 9700
9250 PRINT "restrict ";INDEX.NAME$ : GOTO 9600 'TRY TO CLOSE REMAINING FILES
9260 PRINT "get a new data record" : GOTO 9700
9270 PRINT "read data record #";DRN% : GOTO 9700
9280 PRINT "delete data record #";DATA.RECORD% : GOTO 9700
9290 PRINT "remove old key from ";INDEX.NAME$(KEY%) : GOTO 9500
9300 PRINT "enter key into ";INDEX.NAME$(KEY%) :GOTO 9500
9310 PRINT "write data record #";DRN% : GOTO 9700
9320 PRINT "delete key from ";INDEX.NAME$(KEY%) :GOTO 9500
9330 PRINT "re-establish position"
9332 PRINT "    in ";INDEX.NAME$(KEY%);" after update.": GOTO 9500

9500 CALL CLOSED(FILE.NO%) 'TRY TO CLOSE OTHER INDEX FILES
9510 FOR T.KEY%=0 TO MAX.KEY%
9520    IF T.KEY%<>KEY% THEN CALL RSTRCT(T.KEY%)
9530 NEXT T.KEY%
9540 GOTO 9700 'STOP ERROR MESSAGE

9600 T.KEY%=KEY%+1 'TRY TO CLOSE REMAINING INDEX FILES
9610 IF T.KEY%>MAX.KEY% THEN STOP
9620 FOR KEY%=T.KEY% TO MAX.KEY%
9630    CALL RSTRCT(KEY%)
9640 NEXT KEY%

9700 PRINT
9710 PRINT "DEMONSTRATION TERMINATING WITH ERROR CODE #";ERROR.CODE%
9720 STOP
9900 PRINT 'THESE SHOULD NOT OCCUR. HOWEVER, THEY ARE NOT FATAL.
9910 PRINT "WARNING...Return Code #";RET.CODE%;" occurred while trying to ";
9920 ON ERROR.TYPE% GOTO 9930,9940,9950
9930 PRINT "remove old key from ";INDEX.NAME$(KEY%):GOSUB 18000:RETURN
```

```
9940 PRINT "enter key into ";INDEX.NAME$(KEY%):GOSUB 18000:RETURN
9950 PRINT "delete key from ";INDEX.NAME$(KEY%):GOSUB 18000:RETURN

10000 REM
10010 REM ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
10020 REM
10030 REM                CLEAR SCREEN SUBROUTINE
10040 REM
10050 REM ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
10060 FOR DUMMY%=1 TO 24
10070   PRINT
10080 NEXT DUMMY%
10090 RETURN

11000 REM
11010 REM ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
11020 REM
11030 REM                MAIN MENU SUBROUTINE
11040 REM
11050 REM ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
11060 PRINT TAB(21);"MICRO B+(tm) DEMONSTRATION" :PRINT
11070 PRINT TAB(20);"Customer Database Operations"
11080 PRINT TAB(20);"****************************":PRINT :PRINT
11090 PRINT TAB(5);"1. Enter New Customers"
11100 PRINT TAB(5);"2. Scan/Update/Delete Customer Records"
11110 PRINT TAB(5);"3. List Customer ʀecords"
11120 PRINT TAB(5);"4. Database Statistics"
11130 PRINT TAB(5);"5. Save All Files & Restart Operations"
11140 PRINT TAB(5);"6. Terminate Operations":PRINT :PRINT
11150 INPUT "Enter desired operation number>>",CHOICE%
11160 IF CHOICE%<1 OR CHOICE%>6 THEN PRINT :PRINT :GOTO 11090 ELSE RETURN

12000 REM
12010 REM ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
12020 REM
12030 REM                DATA ENTRY SUBROUTINE
12040 REM
12050 REM ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
12060 IF ENTER.MODE$="NEW" THEN _ 'CLEAR OLD FIELDS IF NEW
        FOR FLD%=0 TO MAX.FIELD%  : _
                OLD.FLD$(FLD%)="" : _
        NEXT FLD%
12070 IF ENTER.MODE$="OLD" THEN _ 'DATA RECORD WILL HAVE BEEN READ
        FOR FLD%=0 TO MAX.FIELD%                : _
                NEW.FLD$(FLD%)=OLD.FLD$(FLD%) : _ 'SET NEW FIELDS
        NEXT FLD%
12080 GOSUB 10000 'CLEAR SCREEN
12090 WHILE ENTER.MODE$="NEW"
12095   PRINT TAB(20);"Enter New Customer Information"
12096   PRINT TAB(20);"****************************":PRINT :PRINT
12097   PRINT TAB(5);"[A zero customer number will terminate input.]":PRINT
```

```
12100   FOR FLD%=0 TO MAX.FIELD%
12110           FLD.NO%=FLD%+1
12120           PRINT TAB(4);FLD.NO%;"- ";FLD.NAME$(FLD%);TAB(30);"("; _
                      FLD.LEN%(FLD%);")";TAB(38);
12130           LINE INPUT ">>",NEW.FLD$(FLD%)
12132           IF FLD%=0 AND VAL(NEW.FLD$(FLD%))=0 THEN _
                      ACTION$="STOP" : _
                      RETURN
12135           NEW.FLD$(FLD%)=LEFT$(NEW.FLD$(FLD%),FLD.LEN%(FLD%))
12140           IF FLD%=0 THEN _
                      GOSUB 20000 : _ 'TEST UNIQUENESS OF CUST #
                      IF NOT UNIQUE% THEN GOTO 12120
12150   NEXT FLD%
12160   ENTER.MODE$="NEWMOD"
12170 WEND

12200 PRINT :PRINT :PRINT
12210 PRINT TAB(20);"Current customer information" : PRINT
12220 FOR FLD%=0 TO MAX.FIELD%
12230   FLD.NO%=FLD%+1
12240   PRINT TAB(4);FLD.NO%;"- ";FLD.NAME$(FLD%);TAB(30);NEW.FLD$(FLD%)
12250 NEXT FLD%
12260 IF ENTER.MODE$="NEWMOD" THEN 12500 'NEW DATA HAS FEWER OPTIONS

12300 PRINT :PRINT
12310 PRINT "Press 'RETURN' to continue scan, enter Field # to change data,"
12320 PRINT "S to save changes, D to delete data, B for back scan, or E"; _
        " to end scan";
12330 LINE INPUT ">>",OP$
12340 IF OP$="" THEN ACTION$="CONT":RETURN
12350 IF OP$="S" OR OP$="s" THEN ACTION$="SAVE":RETURN
12360 IF OP$="D" OR OP$="d" THEN ACTION$="DELT":RETURN
12370 IF OP$="B" OR OP$="b" THEN ACTION$="BACK":RETURN
12380 IF OP$="E" OR OP$="e" THEN ACTION$="STOP":RETURN
12390 OP%=VAL(OP$)
12400 IF OP%<1 OR OP%>NO.FIELDS% THEN 12300
12410 GOSUB 21000 'UPDATE DATA FIELD
12420 GOTO 12200

12500 PRINT :PRINT
12510 PRINT "Press 'RETURN' to save data, enter Field # to change data,"
12520 LINE INPUT "D to delete data, or E to end input>>",OP$
12530 IF OP$="" OR OP$="S" OR OP$="s" THEN ACTION$="SAVE":RETURN
12540 IF OP$="D" OR OP$="d" THEN ACTION$="DELT":RETURN
12550 IF OP$="E" OR OP$="e" THEN ACTION$="STOP":RETURN
12560 OP%=VAL(OP$)
12570 IF OP%<1 OR OP%>NO.FIELDS% THEN 12500
12580 GOSUB 21000 'UPDATE DATA FIELD
12590 GOTO 12200
```

```
13000 REM
13010 REM ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
13020 REM
13030 REM                UPDATE INDICES & DATA FILE SUBROUTINE
13040 REM
13050 REM ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
13060 IF DATA.RECORD%=0 THEN CALL NEWDAT(FILE.NO%,DATA.RECORD%)
13070 IF ERROR.CODE%<>0 THEN ERROR.TYPE%=6:GOTO 9000
13080 DRN%=DATA.RECORD%
13090 FOR KEY%=0 TO MAX.KEY%
13100   FLD%=KEY.MAP%(KEY%)
13110   IF OLD.FLD$(FLD%)<>NEW.FLD$(FLD%) THEN GOSUB 22000 'ADD NEW KEY VALUE
13120 NEXT KEY%
13130 FOR FLD%=0 TO MAX.FIELD%
13140   IF OLD.FLD$(FLD%)<>NEW.FLD$(FLD%) THEN _
            GOSUB 23000: RETURN ' WRITE UPDATED RECORD
13150 NEXT FLD%
13160 RETURN

14000 REM
14010 REM ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
14020 REM
14030 REM                SELECT SEARCH KEY SUBROUTINE
14040 REM
14050 REM ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
14060 GOSUB 10000 'CLEAR SCREEN
14070 PRINT TAB(25);"Customer Database Search Keys":PRINT :PRINT
14080 FOR KEY%=0 TO MAX.KEY%
14090   KEY.NO%=KEY%+1
14100   PRINT TAB(5);KEY.NO%;"- ";KEY.NAME$(KEY%)
14110 NEXT KEY%
14120 PRINT :PRINT
14130 INPUT "Enter desired key number>>",CHOICE%
14140 IF CHOICE%<1 OR CHOICE%>NO.KEYS% THEN 14120
14150 CHOICE%=CHOICE%-1
14160 RETURN

15000 REM
15010 REM ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
15020 REM
15030 REM       CONVERT TARGET VALUE TO KEY FORMAT SUBROUTINE
15040 REM
15050 REM ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
15060 IF KEY.TYPE%(KEY%)=1 THEN _ 'CONVERT TO INTEGER FORMAT
        CONV.TARGET$=MKI$(VAL(TARGET$)) : _
        RETURN
15070 KL%=KEY.LEN%(KEY%)
15080 CONV.TARGET$=LEFT$(TARGET$+SPACE$(KL%),KL%-2)+MKI$(0)
15090 RETURN
```

```
16000 REM
16010 REM ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
16020 REM
16030 REM                READ DATA RECORD SUBROUTINE
16040 REM
16050 REM ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
16060 CALL READD(FILE.NO%,DRN%,BUFFER.PTR%)
16070 IF ERROR.CODE%<>0 THEN ERROR.TYPE%=7:GOTO 9000
16080 OLD.FLD$(0)=MID$(STR$(CVI(CUST.NO$)),2) 'CONVERT CUSTOMER # TO STRING
16090 OLD.FLD$(1)=F.NAME$: OLD.FLD$(2)=L.NAME$: OLD.FLD$(3)=ADDRESS$
16100 OLD.FLD$(4)=CITY$: OLD.FLD$(5)=STATE$: OLD.FLD$(6)=ZIPCODE$
16110 OLD.FLD$(7)=STATUS$
16120 GOSUB 26000 'STRIP TRAILING BLANKS FROM OLD.FLD'S
16130 RETURN

17000 REM
17010 REM ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
17020 REM
17030 REM                DELETE INDEX & DATA FILE ENTRY SUBROUTINE
17040 REM
17050 REM ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
17060 FOR KEY%=0 TO MAX.KEY%
17070   FLD%=KEY.MAP%(KEY%)
17080   IF OLD.FLD$(FLD%)<>"" THEN GOSUB 24000 'DELETE KEY VALUE
17090 NEXT KEY%
17100 CALL RETDAT(FILE.NO%,DATA.RECORD%) 'RETURN DATA RECORD
17110 IF ERROR.CODE%<>0 THEN ERROR.TYPE%=8:GOTO 9000
17120 RETURN

18000 REM
18010 REM ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
18020 REM
18030 REM                PAUSE SUBROUTINE
18040 REM
18050 REM ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
18060 PRINT
18070 LINE INPUT "Press 'RETURN' to continue ---",PAUSE$
18080 RETURN

19000 REM
19010 REM ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
19020 REM
19030 REM                COMPARE INDEX.KEY & U.VALUE SUBROUTINE
19040 REM
19050 REM ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
19060 IF KEY.TYPE%(KEY%)=1 THEN GOTO 19200 'GOTO NUMERIC COMPARE
19070 KL%=KEY.LEN%(KEY%)-2 'ADJUST FOR LAST TWO BYTES (DATA REC #)
19080 C1$=LEFT$(INDEX.KEY$+SPACE$(KL%),KL%)
19090 C2$=LEFT$(U.VALUE$+SPACE$(KL%),KL%)
```

```
22000 REM
22010 REM ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
22020 REM
22030 REM                ADD NEW KEY VALUE SUBROUTINE
22040 REM
22050 REM ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
22055 K.FLD%=KEY.MAP%(KEY%): KL%=KEY.LEN%(KEY%) 'SETUP PARAMETERS
22060 IF KEY%=UNIQ.KEY% THEN 22200 'TRANSFORM TO NUMERIC KEY
22070 SUFFIX$=MKI$(DRN%) 'APPENDED TO NON-NUMERIC KEYS TO MAKE UNIQUE
22080 IF OLD.FLD$(K.FLD%)="" THEN _
        OLD.KEY$="" _
      ELSE _
        OLD.KEY$=LEFT$(OLD.FLD$(K.FLD%)+SPACE$(KL%),KL%-2)+SUFFIX$
22090 IF NEW.FLD$(K.FLD%)="" THEN _
        NEW.KEY$="" _
      ELSE _
        NEW.KEY$=LEFT$(NEW.FLD$(K.FLD%)+SPACE$(KL%),KL%-2)+SUFFIX$
22100 GOTO 22300 'SKIP NUMERIC TRANSFORMATION

22200 IF OLD.FLD$(K.FLD%)="" THEN _
        OLD.KEY$="" _
      ELSE _
        OLD.KEY$=MKI$(VAL(OLD.FLD$(K.FLD%)))
22210 NEW.KEY$=MKI$(VAL(NEW.FLD$(K.FLD%)))

22300 CALL REMOVE(KEY%,OLD.KEY$,DRN%,RET.CODE%) 'REMOVE OLD KEY VALUE
22310 IF ERROR.CODE%<>0 THEN ERROR.TYPE%=9:GOTO 9000
22320 IF RET.CODE%<>1 THEN ERROR.TYPE%=1:GOSUB 9900 'PRINT WARNING
22330 CALL ENTER(KEY%,NEW.KEY$,DRN%,RET.CODE%) 'ADD NEW KEY VALUE
22340 IF ERROR.CODE%<>0 THEN ERROR.TYPE%=10:GOTO 9000
22350 IF RET.CODE%<>1 THEN ERROR.TYPE%=2:GOSUB 9900 'PRINT WARNING
22360 RETURN

23000 REM
23010 REM ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
23020 REM
23030 REM              WRITE NEW DATA RECORD SUBROUTINE
23040 REM
23050 REM ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
23060 LSET CUST.NO$=MKI$(VAL(NEW.FLD$(0)))
23070 LSET F.NAME$=NEW.FLD$(1)
23080 LSET L.NAME$=NEW.FLD$(2)
23090 LSET ADDRESS$=NEW.FLD$(3)
23100 LSET CITY$=NEW.FLD$(4)
23110 LSET STATE$=NEW.FLD$(5)
23120 LSET ZIPCODE$=NEW.FLD$(6)
23130 LSET STATUS$=NEW.FLD$(7)
23140 CALL WRITED(FILE.NO%,DRN%,BUFFER.PTR%)
23150 IF ERROR.CODE%<>0 THEN ERROR.TYPE%=11:GOTO 9000
23160 RETURN
```

```
24000 REM
24010 REM ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
24020 REM
24030 REM                DELETE KEY VALUE FROM INDEX SUBROUTINE
24040 REM
24050 REM ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
24060 K.FLD%=KEY.MAP%(KEY%): KL%=KEY.LEN%(KEY%) 'SETUP PARAMETERS
24070 IF KEY%=UNIQ.KEY% THEN 24200 'TRANSFORM NUMERIC KEY

24080 SUFFIX$=MKI$(DRN%) 'UNIQUE FIELD ADDED TO END OF NON-NUMERIC KEYS
24090 IF OLD.FLD$(K.FLD%)="" THEN _
        OLD.KEY$="" _
      ELSE _
        OLD.KEY$=LEFT$(OLD.FLD$(K.FLD%)+SPACE$(KL%),KL%-2)+SUFFIX$
24100 GOTO 24300 'SKIP NUMERIC TRANSFORMATION

24200 IF OLD.FLD$(K.FLD%)="" THEN _
        OLD.KEY$="" _
      ELSE _
        OLD.KEY$=MKI$(VAL(OLD.FLD$(K.FLD%)))

24300 CALL REMOVE(KEY%,OLD.KEY$,DRN%,RET.CODE%)
24310 IF ERROR.CODE%<>0 THEN ERROR.TYPE%=12:GOTO 9000
24320 IF RET.CODE%<>1 THEN ERROR.TYPE%=3:GOSUB 9900 'PRINT WARNING
24330 RETURN

25000 REM
25010 REM ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
25020 REM
25030 REM                LIST CUSTOMER RECORD SUBROUTINE
25040 REM
25050 REM ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
25060 IF ROUTE$="Y" THEN 25200 'SKIP TO LINE PRINTER ROUTINE
25100 PRINT
25105 PRINT TAB(5);OLD.FLD$(0);TAB(15);OLD.FLD$(7)
25110 PRINT TAB(25);OLD.FLD$(1);" ";OLD.FLD$(2)
25120 PRINT TAB(25);OLD.FLD$(3)
25130 PRINT TAB(25);OLD.FLD$(4);", ";OLD.FLD$(5);"    ";OLD.FLD$(6)
25150 PRINT
25160 RETURN
25200 LPRINT
25205 LPRINT TAB(5);OLD.FLD$(0);TAB(15);OLD.FLD$(7)
25210 LPRINT TAB(25);OLD.FLD$(1);" ";OLD.FLD$(2)
25220 LPRINT TAB(25);OLD.FLD$(3)
25230 LPRINT TAB(25);OLD.FLD$(4);", ";OLD.FLD$(5);"    ";OLD.FLD$(6)
25250 LPRINT
25260 RETURN
```