BIONIC BASIC

by

Glynn Owen

DENVER, CO 80237

Apparat, Inc.

WHAT IS IT ??

BIONIC BASIC is a means of increasing the vocabulary of MICROSOFT's DISK BASIC as supported by the TRS-80 MODEL I computer. This is accomplished by teaching DISK BASIC to recognize new words, and to take the appropriate action each time one of those new words is encountered. The action consists of executing a particular machine language routine which BIONIC BASIC has appended to the BASIC/CMD file. Such an appendage will load along with the other DISK BASIC features each time DISK BASIC is requested by the Disk Operating System.

BIONIC BASIC LIBRARY ROUTINES are appended to a specific copy of the BASIC/CMD file by using the BIONIC BASIC modules. These modules are machine language programs that may be used to customize any particular copy of BASIC/CMD with any or all of the LIBRARY ROUTINES available. There are three BIONIC BASIC modules.

THE FIREUP/CMD MODULE

This module is used to prepare the BASIC/CMD file on the system diskette that is in drive 0 when the module is invoked. Like most other machine language programs, this module is invoked from DOS by entering its name. There are four requirements that must be met before the FIREUP module can be invoked.

1. The FIREUP/CMD program must be on-line.
2. The BIONIC BASIC LIBRARY FILE named CMD/CMD must be on-line.
3. The diskette in drive 0 must have a copy of the BASIC/CMD.BASIC file, and at least 2 free GRANS.
4. The resident Disk Operating System must be the same one that supports the copy of BASIC/CMD found on drive 0.

Once these conditions are met, type the name FIREUP, and hit ENTER. Program operation is automatic from this point on.

When the FIREUP module is finished, it will display this message:

```
STARTUP PACKAGE INSTALLED
```

This is just to let you know that everything went according to plan. Considerable error checking is done during the execution of the FIREUP module. Any errors should be caught before the BASIC/CMD file on drive 0 is altered. If an error does occur, make a note of the message displayed, correct the error, and try again.

Using the FIREUP module on a copy of BASIC/CMD will not make any changes that are visible or useful to the user. This module appends about 200 bytes of code to the BASIC/CMD file that are used by the ROM interpreter to find the LIBRARY ROUTINES when they are installed. No results will be apparent until one or more LIBRARY ROUTINES are installed in this copy of the BASIC/CMD file. Please note that the BASIC/CMD file as specified above must include the standard BASIC password "BASIC".

THE INSTALL/CMD MODULE

As you can guess from its name, this module is used to install LIBRARY FILES. Like FIREUP, it is invoked from DOS by entering its name. The name of the LIBRARY FILE that is being installed must be passed to the INSTALL module. This is done at the same time the module is invoked by following the module name with a space, and then typing the name of the routine. For instance, the LIBRARY FILE named SORT would be installed by entering the command:

```
INSTALL SORT
```

The space between the name of the module and the name of the LIBRARY FILE is mandatory.

Before the INSTALL module is invoked, be sure that:

1. The INSTALL module is on-line.
2. The LIBRARY FILE being installed is on-line.
3. The diskette in drive 0 has a copy of the BASIC/CMD.BASIC file, and at least 2 free GRANS.
4. The resident Disk Operating System is the same one that supports the copy of BASIC/CMD found on drive 0.

Again, please note that the copy of the BASIC/CMD file on drive 0 must include the password "BASIC".

If you try to use the INSTALL module on a copy of BASIC/CMD that has not been operated upon by the FIREUP module, the INSTALL module will tell you so, and abort its run. Likewise, if you try to INSTALL a routine in BASIC that is already there, the module will tell you so, and abort its run.

When the INSTALL module is finished, it will display this message:

```
INSTALLATION COMPLETE
```

This is just to let you know that everything went according to plan. Considerable error checking is done during the execution of the INSTALL module. Any errors should be caught before the BASIC/CMD file on drive 0 is altered. If an error does occur, make a note of the message displayed, correct the error, and try again. It is important to realize that the LIBRARY FILE named to the INSTALL module will actually be appended to the BASIC/CMD.BASIC file on the SYSTEM diskette in Drive 0. If there is not enough room for the expanded BASIC/CMD file, the INSTALL will fail.

THE REMOVE/CMD MODULE

This module is used to remove unwanted BIONIC BASIC LIBRARY FILES from the copy of BASIC/CMD in drive 0. It is invoked in exactly the same format as the INSTALL module. Before using this module, be sure that:

1. The REMOVE module is on-line.
2. The diskette in drive 0 has a copy of the BASIC/CMD.BASIC file.
3. The resident Disk Operating System is the same one that supports the copy of BASIC/CMD found on drive 0.

Again, please note that the copy of the BASIC/CMD file on drive 0 must include the password "BASIC".

When the REMOVE module is finished, it will display this message:

```
REMOVAL COMPLETE
```

This is just to let you know that everything went according to plan. Considerable error checking is done during the execution of the REMOVE module. Any errors should be caught before the BASIC/CMD file on drive 0 is altered. If an error does occur, make a note of the message displayed, correct the error, and try again.

BIONIC BASIC AND THE VARIOUS OPERATING SYSTEMS

All BIONIC BASIC modules and LIBRARY ROUTINES will operate in the environment of a Model I TRS-80 with at least 1 disk drive, and which uses one of the following Disk Operating Systems:

1. NEWDOS 80
2. NEWDOS 2.1
3. VTOS 4.0
4. TRSDOS 2.2
5. TRSDOS 2.3

Attempting to use any of the BIONIC BASIC modules with any other operating system may have unpredictable consequences, although hopefully, any such attempt will abort with this message:

```
TYPE OF DOS NOT RECOGNIZED
```

BIONIC BASIC CONVENTIONS

In an attempt to make the LIBRARY ROUTINES as easy to use as other BASIC abilities, certain conventions have been adopted by the author. These conventions involve the manner in which the LIBRARY ROUTINES are invoked, and the manner in which needed information is passed to these routines.

INVOCATION

LIBRARY ROUTINES are invoked in the following format:

```
CMD NAME , PARAMETER LIST
```

Essentially, this means that all BIONIC BASIC LIBRARY ROUTINES are invoked by using the BASIC keyword CMD. When this keyword is encountered by the BASIC interpreter, the interpreter will search for a LIBRARY ROUTINE name immediately following CMD. If no such name is found, the interpreter will assume that CMD is to be used in its original sense, and it will then attempt to apply it in that fashion. This means that BIONIC BASIC will not interfere with such uses of CMD as CMD "T", or CMD "DIR", etc.

When CMD is followed by the name of any LIBRARY ROUTINE, that routine will be executed, and the interpreter will then move on to its next instruction. This makes it possible to use named LIBRARY ROUTINES in the text of a BASIC program just as though they were legitimate instructions. The routines may also be invoked by name from the keyboard level of operation.

The PARAMETER LIST mentioned in the above format is always required, even if it is empty. As an example, the LIBRARY ROUTINE named COMPRESS may be invoked as:

```
CMD COMPRESS,
```

No parameters are needed, because the information this routine needs to do its job is present at key addresses within the RAM that is used by the ROM interpreter. Notice that the routine name has been followed by a comma to indicate an empty parameter list.

On the other hand, the routine named SORT may have up to three parameters in its parameter list. One sample command for this routine is:

```
CMD SORT, AR(0), "UP", NDX(0)
```

Notice that each parameter uses a COMMA to separate it from its neighbors. The documentation for each routine includes the requirements for the parameter list of that routine.

LINE NUMBERS AND ADDRESSES

Another convention adopted by BIONIC BASIC involves parameters that are line numbers or memory addresses. Such integers must be within the range of 0 thru 65535, instead of of -32768 thru 32767. Such integer expressions are automatically adjusted to this range by BIONIC BASIC. The user does not have to concern himself with whether or not a particular expression is going to cross the boundary at 32767.

The LIBRARY ROUTINE named MOVE uses three such parameters. An example of this command is:

```
CMD MOVE , 15360 , 64000 , 1024
```

This command will MOVE 1024 bytes of memory from 15360 (VIDEO RAM), to 64000 (Protected Memory). If the BIONIC BASIC integer conventions were not in effect, the number 64000 would be misinterpreted as a single precision number. BIONIC BASIC will interpret any expression that evaluates between 0 and 65535 as an integer, when such an expression is being used as a memory address, or as a line number.

Note: All spaces in BIONIC BASIC commands are optional.

BIONIC BASIC LIBRARY ROUTINES AND FILES

The current collection of LIBRARY ROUTINES consists of the following:

1.	FLASH	Turn a flashing cursor on and off
2.	REPEAT	Turn a repeat-key function on and off
3.	WHAT	Display BIONIC BASIC routines
4.	SORT	Sorts any type of list
5.	SEARCH	Find (& replace) specified BASIC text
6.	REDIM	Redimensions one or more arrays
7.	LOOKUP	Find specific data within an array
8.	=	LABEL a routine in a BASIC program
9.	GOTO	Transfer to a LABEL or an expression
10.	GOSUB	Call a LABEL or an expression
11.	RESTORE	Restore to an expression
12.	MOVE	Move a block of memory
13.	LINE	Move a block of program lines
14.	COMPRESS	Remove needless spaces & all remarks
15.	SHIFT	Change the case of a specified string
16.	RUN	RUN, but save the variables
17.	LOAD	LOAD, but save the variables

More routines are scheduled for release in 1981.

These routines exist as relocatable machine language files. Use of the word "relocatable" is not too precise in this application, since the routines are only relocatable by the INSTALL module. Those files are called:

1.	FLASH	Code for FLASH & REPEAT
2.	WHAT	Code for WHAT
3.	SORT	Code for SORT
4.	SEARCH	Code for SEARCH
5.	REDIM	Code for REDIM & LOOKUP
6.	GOSUB	Code for LABEL, GOTO, & GOSUB
7.	RESTORE	Code for RESTORE
8.	MOVE	Code for MOVE
9.	LINE	Code for LINE
10.	COMPRESS	Code for COMPRESS
11.	SHIFT	Code for SHIFT
12.	LOAD	Code for RUN & LOAD

It is important to know that these file names MUST NOT BE CHANGED. Any exceptions to this rule will be noted in the documentation for the routine involved. Some of the files contain more than one routine. Such routines are all INSTALLED or REMOVED at the same time. The GOSUB file is a good example. This file is installed by entering the command:

```
INSTALL GOSUB
```

This command will append the code for the LABEL, GOTO, and GOSUB enhancements. After this code has been appended, the WHAT command will display the presence of all three of these commands. If this command is issued from DOS:

```
REMOVE GOSUB
```

All three routines will be removed.

While the GOSUB routine is a part of DISK BASIC, you might be tempted to try a:

```
REMOVE GOTO
```

The REMOVE module will tell you that the routine has not yet been installed, even though the WHAT command will tell you that it is there. Multiple routines in a single file are installed and removed under the name of the first routine in the file. The other routines are accessible by the BASIC interpreter, but the REMOVE module will not attempt to take them out.

If you are using VTOS 4.0, you should know that a BASIC/CMD file that has been altered by BIONIC BASIC must not be used with the PATCH utility.

If this is your first time through the installation of a BIONIC BASIC routine, you should realize that the following steps will permanently alter the BASIC/CMD file that is on the SYSTEM diskette in drive 0. The BASIC/CMD file is normally invisible, so you may not be aware of it. It is always a good idea to work with a copy of the SYSTEM diskette first, and then do a BACKUP if everything goes as it should. This will guarantee that you do not lose valuable files. This is particularly true if you only have one drive.

If you only have one drive, use the LOADER on the Distribution Diskette to bring the STARTER package on-line. If you have more than 1 drive, put a copy of the Distribution Diskette into Drive 1. The first thing you must do is verify that your SYSTEM diskette has 10 free GRANS. These GRANS will be allocated to the BASIC/CMD file on Drive 0 as the BIONIC BASIC modules append code to it. If you have used the LOADER and followed the instructions for it, your SYSTEM diskette will have enough room on it.

Once you are sure that the SYSTEM diskette has enough space free, use the FIREUP module to condition the BASIC/CMD file on your SYSTEM diskette. With the DOS READY prompt showing, Type the word FIREUP and push ENTER. The FIREUP module will load and run. When it is finished, it will display this message:

```
STARTUP PACKAGE INSTALLED
```

The STARTUP package is a 200 byte appendage to the BASIC/CMD file which will enable the ROM interpreter to find and execute the BIONIC BASIC LIBRARY ROUTINES once they have been added. All by itself, this code does nothing but use memory. One or more LIBRARY ROUTINES must be added before the capabilities of the STARTUP package become useful.

Once the STARTUP package has been appended to the BASIC/CMD file on a particular SYSTEM diskette, you may add any of the BIONIC BASIC LIBRARY FILES to Disk Basic when that diskette (or a copy of it) is being used as the SYSTEM diskette in Drive 0. You may also copy the altered BASIC/CMD file over to a different SYSTEM diskette, and the same thing will still be true. You must be certain that there is enough room on the diskette for any additions that you intend to make.

Each LIBRARY FILE contains one or more of the LIBRARY ROUTINES of BIONIC BASIC. One of the most useful of the LIBRARY FILES in the STARTER package is named GOSUB. This file contains the machine language code that will enable Disk Basic to use computed and labelled GOTOs and GOSUBs. To install these capabilities, simply type:

```
INSTALL GOSUB
```

and hit ENTER. The INSTALL/CMD module will append the code in the GOSUB file to the BASIC/CMD file on drive 0 so that the GOSUB capabilities will load and execute with all of the other Disk Basic enhancements to Level II.

Another very handy LIBRARY FILE is the one named WHAT. This file contains the code which will enable you to request a display of the currently installed BIONIC BASIC LIBRARY ROUTINES. This file is installed by the command:

```
INSTALL WHAT
```

When the INSTALL module comes back with the message:

```
INSTALLATION COMPLETE
```

Type BASIC and hit ENTER. When you get to Disk Basic, notice the new banner that is on the display. This message is displayed to indicate that you are using a copy of BASIC/CMD that includes the STARTUP package of BIONIC BASIC.

While you are in Disk Basic, ENTER this command:

```
CMD WHAT,
```

You will see the BIONIC BASIC LIBRARY ROUTINES that you have just installed. Notice that there are more LIBRARY ROUTINES installed than there are LIBRARY FILES. Some LIBRARY FILES contain more than one routine. The GOSUB file contains 3.

ENTER a CMD "S" to return to DOS READY.

Each of the other LIBRARY FILES that are part of the STARTER package may now be added to the BASIC/CMD file on Drive 0 by using the INSTALL module. These files are named:

LOAD
RESTORE
MOVE
SHIFT
FLASH

When you have installed all of these files, enter Disk Basic, and RUN the DEMO program that is part of the STARTER package. The DEMO will show you a use for each of the LIBRARY ROUTINES.

The LIBRARY ROUTINES that have been appended to Disk Basic by the INSTALL module can be taken out of the BASIC/CMD file by using the REMOVE module. To try this out, return to DOS READY after you are finished with the DEMO program by entering a CMD "S". Then ENTER this command:

```
REMOVE GOSUB
```

When the REMOVE module comes back with the message:

```
REMOVAL COMPLETE
```

Reenter Disk Basic, and try to RUN the DEMO program. You will find that it will not RUN, since the DEMO program was written using computed and labelled GOTOs and GOSUBs.

Return to DOS READY and ENTER the command:

```
INSTALL GOSUB
```

The GOSUB file will be put back into the BASIC/CMD file. When you get into Disk Basic and use the WHAT command, you will see that the LIBRARY ROUTINES in the GOSUB file are displayed at a different location. The INSTALL module relocates the LIBRARY FILES to fit into memory at the current end of the memory space used by the BASIC/CMD file. Removing the GOSUB file and then reinstalling it caused it to be located in a different segment of memory.

Now that you have an existing copy of the BASIC/CMD file which has been modified to include the BIONIC BASIC LIBRARY ROUTINES, you may copy this file onto other diskettes so that you can use the LIBRARY ROUTINES that you have installed. Remember that the BASIC/CMD file has been expanded to include these routines. Before you try to move this file onto another diskette, be certain that there is room for it. The REMOVE and INSTALL modules will work on any copy of the BASIC/CMD file that includes the STARTUP package.

FILENAME GOSUB INSTALL GOSUB REMOVE GOSUB

COMMAND FORMATS

```
CMD = , "NAME OF ROUTINE"
CMD GOSUB , DESTINATION
CMD GOTO , DESTINATION
```

All three of these routines are contained in the file named GOSUB.

PARAMETER DISCUSSION

CMD = , may be used as a LABEL inside of a BASIC program to mark the beginning of a named routine. The "NAME OF ROUTINE" parameter must always be a literal string. No variables can be used here.

When CMD = , is encountered by the BASIC interpreter, the interpreter simply moves ahead to the next instruction, effectively ignoring the label. The label is not supposed to execute. It is just supposed to tag the beginning of a particular block of code within a BASIC program for access by the GOTO and GOSUB commands.

The DESTINATION parameter for the GOSUB and GOTO commands may be any type of expression. If it is numeric, the line number it represents will be located, and program execution will continue at the first statement on that line. If it is a string, the LABEL routine will search the program line by line for the first line that contains a label that matches the string. Execution will continue at the first statement past the matching label.

If the DESTINATION is not found, an error results.

EXAMPLES OF LABELLED/COMPUTED GOTO & GOSUB

```
10 CMD GOSUB , RND(5) + 19 : PRINT A$ : GOTO 10
20 A$ = "LINE 20" : RETURN
21 A$ = "LINE 21" : RETURN
22 A$ = "LINE 22" : RETURN
23 A$ = "LINE 23" : RETURN
24 A$ = "LINE 24" : RETURN
```

This is an illustration of the computed GOSUB. It will output random line numbers between 20 and 24.

```
10 CMD GOSUB , "MESSAGE" : STOP
20 CMD =, "MESSAGE" : PRINT "BIG DEAL" : RETURN
```

This example will print "BIG DEAL", and then stop.

```
10 I = 0 : CMD =, "TRICK" : I = I + 1 : PRINT I :
   CMD GOTO , "TRICK"
```

This one-liner will print the numbers 1 thru N, where N is some measure of the operators patience. It is included to demonstrate the fact that labels may be used at any point in a program, including the middle of a line.

```
10 A$ = "EXIT"
20 CMD GOTO , A$
30 CMD =, "EXIT" : PRINT "THAT'S ALL FOLKS" : END
```

This illustrates the fact that labels may be ACCESSED using string variables. They may not be DEFINED by string variables.

FILENAME SHIFT   INSTALL SHIFT   REMOVE SHIFT

COMMAND FORMAT

```
CMD SHIFT,STRING1,STRING2
```

PARAMETER DISCUSSION

This command can be used to alter the case of alphabetic characters in STRING1, which may be any literal or variable string. STRING2 is optional. If it is used, it must be a string that begins with "U" (for UPPER case), "L" (for LOWER case), or "S" (for SWITCH case).

EXAMPLES OF THE SHIFT COMMAND

```
CMD SHIFT , A$ , "U"
```

This command will guarantee that every alphabetic character in A$ is upper case.

```
CMD SHIFT , A$ , "L"
```

This command will guarantee that every alphabetic character in A$ is lower case.

```
CMD SHIFT , A$ , "S"
```

This command will switch lower case letters to upper case, and upper case letters to lower case.

```
CMD SHIFT , A$
```

This command is equivalent to example 1.

FILENAME LOAD   INSTALL LOAD   REMOVE LOAD

COMMAND FORMATS

```
CMD LOAD , OPTION1 , VARIABLE LIST
CMD RUN , OPTION1 , VARIABLE LIST, OPTION2
```

Both routines are contained within the library file named LOAD.

PARAMETER DISCUSSION

The LOAD command is used to LOAD a BASIC program into memory from a disk file without changing the variable list that is already in memory, or without losing specified variables. "FILENAME" must be the filename of a BASIC program in standard format. The quotes must be included.

OPTION1 may be any literal number. If such a number is found immediately following the "FILENAME", that amount of string storage space will be reserved after the new program is loaded. If OPTION1 is not used, the amount of string storage space will not be changed from its current value.

The VARIABLE LIST may be 1 or more variable names separated by commas. Such a list is called an "explicit" variable list. This is an example of an explicit variable list :

```
V1, V2, V3, V4
```

Any variable name may be used in an explicit variable list. See example 4 for restrictions on this type of list.

The VARIABLE LIST may consist of a the "*" symbol. This symbol is used in place of an explicit list to indicate that all variables are to be saved. This type of list is called an "implicit" variable list.

OPTION2 may be any line number of PROGRAM2. When OPTION2 is included in the parameter list of CMD RUN, PROGRAM2 will begin its RUN at this line.

EXAMPLES OF CMD LOAD

```
CMD LOAD, "PROGRAM2", *
```

This example will LOAD the program named PROGRAM2 without changing any of the current variables, or altering the amount of string space that has already been reserved.

```
CMD LOAD, "PROGRAM2", 1000, *
```

This example will LOAD PROGRAM2 without changing any of the current variables. After the program is loaded, 1000 bytes of memory will be reserved for string storage.

```
CMD LOAD, "PROGRAM2", I, J, K, L
```

This example will save the variables named I, J, K, and L, and LOAD the program named "PROGRAM2". PROGRAM2 will have the same amount of string space reserved that the original program had.

```
CMD LOAD, "PROGRAM2", I, J, A1(0), A2(0,0), A3(0,0,0)
```

This example will save the scalar variables named I and J, along with the ARRAY variables named A1, A2, and A3 before loading PROGRAM2. There are two things you should notice about this example.

1. SCALARS MUST PRECEDE ARRAY VARIABLES
2. ARRAYS MUST USE ALL ZERO SUBSCRIPTS

If a scalar variable is found after any array has been listed, a TYPE MISMATCH error will occur. If an array is named that does not use the zero subscript, a SUBSCRIPT OUT OF RANGE error will occur.

EXAMPLES OF CMD RUN

```
CMD RUN , "PROGRAM2" , *
```

This command will save the variables from PROGRAM1, load PROGRAM2 into memory, put back the PROGRAM1 variables, and then begin execution of PROGRAM2 at its very first line.

```
CMD RUN , "PROGRAM2" , * , 100
```

This example is identical to the first example, except that OPTION2 has been used to cause PROGRAM2 to begin executing at the start of line number 100.

```
CMD RUN , "PROGRAM2" , A$ , ARRAY(0)
```

This command will save the pointers for A$, and the contents of ARRAY, load PROGRAM2 into memory, and begin execution of PROGRAM2 at the first line.

```
CMD RUN , "PROGRAM2" , A$ , ARRAY(0) , 100
```

This command is identical to example 3, except that OPTION2 has been used to cause PROGRAM2 to begin execution at line number 100.

```
CMD RUN , "PROGRAM2" , 2000 , *
```

This command is identical to example 1, except that PROGRAM2 will begin to RUN with 2000 bytes of memory reserved for string storage.

```
CMD RUN , "PROGRAM2" , 100 , X , Y , Z
```

This command will save the variables named X, Y, and Z, load PROGRAM2 into memory, put back the variables X, Y, and Z; reserve 100 bytes of memory for string storage, and begin executing PROGRAM2 at its very first line.

```
CMD RUN , "PROGRAM2" , 1000 , X , Y , Z , 100
```

This command is identical to example 6, except that OPTION2 has been used to force PROGRAM2 to begin execution at line number 100.

THINGS TO KNOW ABOUT LOAD & RUN

(1) String variables passed by the LOAD (or RUN) command may not come through with their content preserved if the amount of string space reserved for PROGRAM2 is smaller than the amount reserved by PROGRAM1. Strings inside of file buffers are not affected by CMD LOAD or CMD RUN.

(2) Strings that have been assigned to text within PROGRAM1 will not be valid in PROGRAM2. There are 2 kinds of strings like this :

1. READ A$ (assignment from DATA statements)
2. A$ = "JOHN JONES" (literal assignments)

PROGRAM2 will probably overwrite the memory used by the text of these strings. Even though the string pointers are preserved, the content of the strings will not be predictable. Either or both of the example strings could be guaranteed by using

```
A$ = LEFT$(A$,255)
```

in PROGRAM1 before PROGRAM2 is requested.

(3) User-defined functions (DEF FN or DEF USR) will never carry over from one program to another. Such functions must be redefined in each program.

(4) The variable typing table established by the use of DEFINT, DEFSNG, DEFDBL, or DEFSTR is carried over from one program to the next. This may cause a lot of "TYPE MISMATCH" errors if it is

forgotten.

(5) OPEN files in PROGRAM1 will remain OPEN to PROGRAM2. BIONIC BASIC LOAD and RUN will not CLOSE files.

(6) Any error encountered during the physical LOAD of PROGRAM2 by DOS is a fatal error. BASIC program text is erased from memory before the error is reported. This should never be anything but an OUT OF MEMORY error, although other errors are possible.

(7) If no VARIABLE LIST is found, a MISSING OPERAND error will be generated. These commands should not be used in place of the normal LOAD and RUN commands.

(8) All scalar (non-array) variables must be named in an explicit variable list before any array is named. Listing a scalar after the first array has been listed will cause a TYPE MISMATCH error.

(9) Array variables named in an explicit list must include zero subscripts. Non-zero subscripts will cause a SUBSCRIPT OUT OF RANGE error.

FILENAME WHAT INSTALL WHAT REMOVE WHAT

COMMAND FORMAT

```
CMD WHAT,D
```

PARAMETER DISCUSSION

This command will clear the screen and display the names of those BIONIC BASIC routines that are presently available. The "D" parameter is not needed, but the comma after the WHAT is.

FILENAME RESTORE INSTALL RESTORE REMOVE RESTORE

COMMAND FORMAT

```
CMD RESTORE , LINE NUMBER
```

PARAMETER DISCUSSION

The "LINE NUMBER" parameter may be any NUMERIC EXPRESSION. If the integer part of the expression is not an existing line number, an UNDEFINED LINE # error will result. This command allows the programmer the option of placing the internal DATA pointer at any existing line number. This is a very powerful capability for games programs and self-contained databases.

EXAMPLES OF CMD RESTORE

```
CMD RESTORE , 100
```

The DATA pointer will be restored to the beginning of line 100. If line 100 does not exist, an error will be generated.

```
CMD RESTORE , X
```

The DATA pointer will be restored to the line with a number that is equal to the integer part of the number contained in the variable named "X". If no such line exists, an error will be generated.

```
CMD RESTORE , X/5 + 2
```

The DATA pointer will be restored to the line with a number equal to the integer part of the expression "X/5 + 2". If no such line exists, an error will be generated.

FILENAME MOVE INSTALL MOVE REMOVE MOVE

COMMAND FORMAT

```
CMD MOVE , FROM , TO , LENGTH
```

PARAMETER DISCUSSION

All three parameters are required. The FROM parameter is the address of the first byte to be MOVEd by the command, and the TO parameter is the address to which it is MOVEd. The LENGTH parameter specifies how many bytes are to be MOVEd. All three parameters may be any numeric expression that the BASIC interpreter can evaluate. The MOVE command uses the LDIR and LDDR instructions of the Z-80. It can relocate 10,000 bytes of memory in less than 20 milliseconds.

This command has a variety of uses. It can be used to scroll any part of the screen, save the contents of the video display in protected memory, or to save variable contents in protected memory while one program is being replaced with another. This one-line program will scroll the eighth through the twelfth lines of the video display :

```
1000 CMD MOVE,15808,16128,64:
     CMD MOVE,15872,15808,320
```

The thirteenth line is used as a buffer for line 8. An arbitrary string could be used almost as easily.

FILENAME FLASH INSTALL FLASH REMOVE FLASH

COMMAND FORMAT

```
CMD FLASH , OFF/ON , OPTION
CMD REPEAT , OFF/ON , OPTION
```

Both of these enhancements are contained in the library file named FLASH.

PARAMETER DISCUSSION

The OFF/ON parameter may be any variable or literal string that is equal to one of the two words OFF or ON.

When the OPTION is used with the FLASH command, it determines the character that is flashed on and off. This parameter defaults to a small graphics block. If the OPTION given is a string expression, then the first character of that string will be FLASHed. If the OPTION is numeric, then CHR$(OPTION) will be FLASHed. The alternating character for the FLASH command will always be the regular underscore cursor. If OPTION=128, then the underscore will blink on and off.

When the OPTION is used with the REPEAT command, it must be a numeric expression. The value of this expression will determine the speed with which a key that is being held down will be accepted as new input. The default value for this parameter is 4000. Any value between 0 and 65535 will be accepted.

Once the OPTION parameter has been used with either the FLASH or REPEAT commands, the default value becomes the OPTION.

EXAMPLES OF FLASH & REPEAT

```
CMD FLASH , "ON" , "#"
```

This will cause the pound sign to alternate with the underscore cursor.

```
CMD FLASH, "OFF"
```

This will turn the flashing cursor off

```
CMD FLASH , "ON" , 128
```

This will cause the underscore cursor to alternate with a graphics blank. Effectively, it will cause the underscore to blink on and off. This is an unobtrusive blinking cursor.

```
CMD REPEAT , "ON"
```

This turns on the repeat key function with the default value for repetition rate.

```
CMD REPEAT , "OFF"
```

This turns off the repeat key function

```
CMD REPEAT , "ON" , 1
```

This turns on the repeat key function, and sets the repetition at the fastest possible rate.

DEMO

The DEMO program is a BASIC program containing subroutines that illustrate most of the currently available library routines. As more routines are added to the library, the programs that demonstrate them will be written to MERGE with this DEMO. Study of these routines should be sufficient to clarify the documentation.

The user must INSTALL both the WHAT and the GOSUB library routines, before the DEMO program will RUN. DEMO uses CMD WHAT to find out which routines have been installed. The CMD GOSUB routine is used by DEMO in almost every subroutine call that it makes.

Several of the subroutines in this program use the STOP command to halt program execution. This causes the ROM interpreter to come back to READY with this message:

```
BREAK IN LINE XXX
READY
```

XXX represents the line number in which the STOP command was encountered. Any time that you see this "BREAK" message, you can resume execution of the DEMO program by entering the command:

```
CONT
```

DEMO uses the STOP command to give the operator a chance to list program lines for study.

If the program ceases execution without printing the "BREAK" message, the operator must enter a GOTO or a RUN command in order to resume execution of the DEMO program. In the original DEMO program, this happens when CMD WHAT is used to display the names of currently installed routines, and when CMD COMPRESS is called upon to execute. To recover from the halt caused by CMD WHAT, enter a:

```
RUN 100
```

It is not possible to recover from the halt caused by CMD COMPRESS. This routine automatically does a CLEAR, so DEMO must be RUN again in order to reinitialize its variables.

The DEMO program may be compressed, but it may not be renumbered. Computed GOSUBs are demonstrated by this program using assigned line numbers. If you change the line numbers, the computed GOSUBs will be off.

The routine that is used to demonstrate the LOAD and RUN commands will move the DEMO program out of and back into memory while preserving its variables. DO NOT use any command while DEMO2 is resident that will alter the variable table.

SYSTEM DISKETTE:

A SYSTEM diskette is one that is programmed to work when it is used in drive 0. The programming on such diskettes includes the Disk Operating System that is necessary before diskette files can be accessed. SYSTEM diskettes used by BIONIC BASIC must include a copy of the BASIC/CMD.BASIC file that is standard with every operating system recognized by BIONIC BASIC.

DISTRIBUTION DISKETTE:
This is the diskette that you purchased. It contains the BIONIC BASIC library files and/or modules.

EXPRESSION:
An expression is a string of ASCII symbols that can be evaluated by the INTERPRETER as a unique number or string.

FILE:
Information that has been stored on a diskette by name is called a "file". Files may store any kind of information.

ON-LINE:
Any file that is on a diskette in a currently active disk drive is said to be "on-line". This means that the Disk Operating System can access the file.

MODULE:
A module is a BIONIC BASIC file that contains a functional machine language program designed to alter the BASIC/CMD.BASIC file on drive 0.

LIBRARY ROUTINE:
A library routine is any one of the enhancements to DISK BASIC that is implemented by BIONIC BASIC.

LIBRARY FILE:
Any file on a DISTRIBUTION diskette that does not include the "/CMD" extension is a LIBRARY FILE. Such files contain one or more of the LIBRARY ROUTINES.

SUBSCRIPT:
Each dimension of an array is called a SUBSCRIPT. An array that is dimensioned as ARRAY(23) has only 1 SUBSCRIPT. That SUBSCRIPT may be any number between 0 and 23, inclusive. Similarly, an array that is dimensioned as ARRAY(X,Y,Z) has 3 SUBSCRIPTS.

LIST:
A LIST is an array that uses only one SUBSCRIPT.

SUBLIST:
A SUBLIST is a part of a multi-dimensional array. A SUBLIST is defined when one SUBSCRIPT is unknown, and all others are valid numeric EXPRESSIONS.

INDEX:
An INDEX is a specific integer value for some SUBSCRIPT of an array.

MARKER:
A symbol that is used to indicate which SUBSCRIPT of an array is unknown.

ROM:
An acronym for Read Only Memory. Level II BASIC is stored in 2 or 3 ROM chips.

RAM:
An acronym for Random Access Memory. A more accurate acronym might be RAW (for Read And Write). The Disk Operating System, Disk Basic, BIONIC BASIC, and all BASIC programs are stored in RAM.

INTERPRETER:
Instructions to the computer that are written in BASIC must be translated into machine language. A large segment of ROM contains a machine language program that performs this task. That program is called an INTERPRETER.

LOADER:
A special program that is loaded into RAM whenever the Distribution Diskette is booted up in Drive 0. The LOADER will transfer the files on the Distribution Diskette onto a SYSTEM diskette.

INVOKE:
When a program (or LIBRARY ROUTINE) is made to execute, it is said to be INVOKED.

PARAMETER:
When a LIBRARY ROUTINE is invoked, it may require one or more items of information before it can properly execute. Those items are referred to as PARAMETERS.

FILENAME SEARCH INSTALL SEARCH REMOVE SEARCH

BASIC TOKENS

Some of the discussion of this command assumes an understanding on the part of the reader of the concept of TOKENS. A token is a one byte representation of an entire LEVEL II BASIC keyword. A list of these keywords is found in the LEVEL II BASIC reference manual on page A/15. Each of these keywords is associated with a single-byte token. When a command is issued from the command mode of BASIC, or when a line of BASIC program is entered, the text of the entry is searched for BASIC keywords by the BASIC interpreter, and every occurrence of a keyword is replaced by that keyword's token.

All BASIC programs are stored in memory with keywords in their tokenized form. When the operator calls for a LISTing of the program, the tokens are recognized by the interpreter, and displayed in their expanded format as a keyword. When a BASIC program is saved to disk by a: SAVE "FILENAME": command, the body of the BASIC program is SAVEd with the keywords in their tokenized form in order to conserve disk space. Issuing the command to: SAVE "FILENAME",A: causes the interpreter to recognize tokens, and save them as the ASCII text of the keyword instead of as tokens. When such a file is called back into memory from the disk, the keywords are tokenized before they are stored.

WHAT DOES SEARCH DO ??

The SEARCH command is an extension of the EDITing capabilities already written into the BASIC language. This command will enable you to SEARCH any BASIC program resident in RAM for a string, and to replace it with some other string. Because there are no restrictions on the strings that may be used for replacement, it is possible for this command to create BASIC program lines that are too long for the BASIC monitor to save or run. Replacements should be chosen with that thought in mind. When no replacement is made, or when the replacement string is exactly the same length as the object string, the exit from the routine is the same as for any other BASIC command. Under such circumstances, this command may be used inside of a program. If a replacement is called for that is different in length from the object string, the routine will exit to the BASIC ready prompt.

Since the variable names used in a program consist of ASCII strings, it is possible for this command to do a global SEARCH & replace of any variable name with any other. It is also possible to find and replace any specified BASIC token with any other, such as changing all PRINT statements to LPRINT. The range of the SEARCH is specified by an upper and lower line number limit. The command searches program text from the lower limit through the upper limit for all occurrences of the object string, and does a replacement of that string if anything appears in the replacement option. Strings that contain a ":" are difficult to find and to replace.

COMMAND FORMAT

```
CMD SEARCH, LOWER, UPPER, OBJECT, REPLACEMENT
```

PARAMETER DISCUSSION

The LOWER and UPPER parameters are numeric. The other two must be literal strings. Either one or both of the numeric parameters may be omitted, although the commas used to separate them must always be present. If no replacement is listed, then the command will simply display the line numbers of each line that contains the object string. Any line containing that string more than once will be printed that many times. Please note that placing a comma after the object string is sufficient for the command to assume that a replacement string is intended, but that the replacement is blank. This means that placing a comma after the object string will cause the command to delete the object string every time it is located.

In order to accomodate the possibility of locating and changing BASIC key words, it is not always mandatory to use quotes around the strings in this command. The BASIC monitor reduces all unquoted BASIC key words to one byte tokens before it executes the SEARCH command. As a result of this fact, it is important to bear in mind what you are looking for before you call for a SEARCH. If what you are looking for is on the list of BASIC key words in the LEVEI II manual, then it must be entered without quotes. Similarly, if you want to replace a string with a key word on the list, then the replacement string should not be quoted. Any other string used as the object or replacement string should be quoted. The BASIC decoding scheme also makes it difficult to replace the DATA and REM tokens with anything other than a literal string.

The UPPER and LOWER parameters may be used to restrict the SEARCH command to a particular segment of the resident program. If the LOWER limit is used, then the SEARCH will begin at the first line of the program with a line number equal to or greater than the LOWER parameter. If it is not used, then the SEARCH defaults to the beginning of the program. The UPPER limit may be any number greater than or equal to the LOWER limit. If the UPPER and LOWER limits are equal, then just that one line is SEARCHed. If the UPPER limit is less than the LOWER limit, then no SEARCH is executed. If no UPPER limit is specified, then the program is SEARCHed from the LOWER limit through to the end of program text.

The SEARCH command will display the line number of every line that contains a match to the object string. This output may be suppressed completely, or routed to a line printer. The command

```
CMD SEARCH,"OFF"
```

will disable all output by this routine. CMD SEARCH,"ON" will restore the display capabilities. The display can be routed to the line printer by: CMD SEARCH,"ON",L. Either of the following commands will route the display to the video:

```
CMD SEARCH,"ON",D
```

CMD SEARCH,"OFF",D

The display switch has been provided to make it possible to use this command inside a program without getting strange output on the screen.

The routine was designed for use in EDITing resident BASIC programs. The programmer may use the routine to locate specific strings. Once found, the strings may be replaced by the routine, or by using the EDIT commands of the computer. Here are some examples of how this routine may be used in EDITing a program. The examples assume that the routine has been initialized with a---CMD SEARCH,"ON",D---so that output from SEARCH will go to the video display.

EXAMPLES OF COMMAND SEARCH

EXAMPLE (1) CMD SEARCH,,,"A$"

This command will SEARCH the entire program, and display the line number of each line that contains any string equal to A$. Please note that the routine will report the variable RA$ as an occurrence of A$.

EXAMPLE (2) CMD SEARCH,100,1000,"A$"

This command will do exactly the same thing as example (1), except that the routine will not begin its SEARCH until it finds the first line number greater than or equal to 100. The SEARCH will cease the first time the routine encounters a line number greater than 1000.

EXAMPLE (3) CMD SEARCH,100,,"A$"

This command is identical to EXAMPLE (2) except that program text will be SEARCHed from line number 100 through to the end of the program.

EXAMPLE (4) CMD SEARCH,,1000,"A$"

This command is identical to EXAMPLE (1) except that the SEARCH will be made from the first line number (as long as it is less than 1000) on up to line number 1000.

EXAMPLE (5) CMD SEARCH,,,REM

This command will SEARCH the entire program, and display the line numbers of every line that contains a REMark.

EXAMPLE (6) CMD SEARCH,,,"REM"

This command will SEARCH the entire program, and display the line numbers of every line that contains the ASCII characters "REM".

EXAMPLE (7) CMD SEARCH,,,"A$","A1$"

This command is identical to EXAMPLE (1) except that every time the routine locates A$, it will replace it with A1$.

EXAMPLE (8) CMD SEARCH,,,"A$",

This command is identical to EXAMPLE (1) except every occurrence of A$ will be deleted from the program. Please note that the only difference between this command and the one given in EXAMPLE (1) is the comma after the A$.

```
     EXAMPLE (9) CMD SEARCH,,,"A$",""
```

This command is identical to EXAMPLE (8).

```
     EXAMPLE (10) CMD SEARCH,,,DATA,REM
```

This command appears to be a way to change DATA statements to REMarks. It will not work as intended. The BASIC interpreter sees every character in this command beyond the keyword DATA as part of a DATA statement. This means that every character is interpreted literally, so that the keyword REM will not be tokenized. The result of this command will replace every token of the keyword DATA with the literal string "REM".

In order to accomplish the replacement of the DATA token with the REM token it would be necessary to do a 2-part replacement like so:

```
     CMD SEARCH,,,DATA,"XYZ" : CMD SEARCH,,,"XYZ",REM
```

This discussion should make it clear that no keyword in a replacement will be correctly tokenized if it follows either of the two keywords REM or DATA.

FILENAME COMPRESS   INSTALL COMPRESS REMOVE COMPRESS

COMMAND FORMAT

```
CMD COMPRESS , OPTION
```

PARAMETER DISCUSSION

The optional parameter of this command serves no useful purpose. It is only there because the comma after the word COMPRESS is mandatory. This command exits to the READY mode, and should only be used as a KEYBOARD command, in the same way that EDIT is used as a KEYBOARD command. Once a program has been COMPRESSed, it may be SAVEd, LOADed, and RUN as usual. On large programs, this command may save thousands of bytes of memory.

FILENAME LINE   INSTALL LINE   REMOVE LINE

COMMAND FORMAT

```
CMD LINE, SOURCE : DESTINATION : INCREMENT : FLAG
```

Notice that the terminators between parameters of this command are colons, except for the comma between the name "LINE", and the parameter which specifies the SOURCE for the move.

PARAMETER DISCUSSION

The "SOURCE" parameter specifies the block of lines that is to be moved. If L1 & L2 are 2 different line numbers, then the SOURCE parameter may take on either of these formats:

```
     L1
     L1 - L2
```

The first format of the SOURCE parameter will cause only the one line numbered L1 to be moved. The second will move the block of lines between and including L1 & L2. Any line number used in the SOURCE parameter must be an EXISTING LINE NUMBER, or no move will occur.

The DESTINATION parameter specifies the first new line number that will be used by the lines being moved from the SOURCE. Only one line number is needed here. It MUST NOT BE AN EXISTING LINE NUMBER. If it exists, an error will be generated that reads:

```
OVERFLOW INTO EXISTING LINE #
CAN'T CONTINUE
```

and the move will be aborted.

The "INCREMENT" parameter is used to specify the increment between line numbers at the DESTINATION whenever more than one line is moved from the SOURCE. This parameter is optional. It will default to 5 if not specified. Whenever more than one line is to be moved, the LINE routine inspects the resident program for any line number that is less than or equal to the largest line number that will result from adding the INCREMENT to the DESTINATION line number for the number of lines that are being moved. If such a line does exist, then the error message above will be displayed, and the move will be aborted.

The "FLAG" parameter has one possibility, which is the letter "D", without quotes. When a D appears in this position, the LINE routine will delete the original lines after moving them to their new location. Otherwise, they are not changed.

VALID POSSIBILITIES FOR THE "LINE" COMMAND

If L1, L2, and L3 are three different line numbers, then the following are valid commands:

```
CMD LINE, L1 : L2
```

Will duplicate L1 at L2. L1 will still exist.

```
CMD LINE, L1 : L2 : D
```

Will move L1 to L2 and delete L1.

```
CMD LINE, L1-L2 : L3
```

Will move lines L1 thru L2 into the line numbers that begin with L3 and INCREMENT in the default steps of 5.

```
CMD LINE, L1-L2 : L3 : D
```

This does exactly the same thing as the previous example, except that lines L1 thru L2 are deleted after they are moved.

```
CMD LINE, L1-L2 : L3 : 1
```

Will move lines L1 thru L2 into the line numbers that begin with L3 and go up in steps of 1.

```
CMD LINE, L1-L2 : L3 : 1 : D
```

Will do exactly the same thing as the previous example, except that the SOURCE lines will be deleted. Notice that the INCREMENT parameter precedes the deletion FLAG.

UNLIKE OTHER BIONIC BASIC ROUTINES THAT ARE LINE NUMBER ORIENTED, THIS ROUTINE REQUIRES LITERAL LINE NUMBERS SUCH AS 10, 123, 1014, ETC. VARIABLES AND EXPRESSIONS MUST NOT BE USED.

FILENAME SORT INSTALL SORT REMOVE SORT

COMMAND FORMAT

```
CMD SORT , ARRAY , DIRECTION , INDEX
```

PARAMETER DISCUSSION

The SORT command may be used to put the elements of the ARRAY parameter into ascending or descending order. The ARRAY parameter may be the zero index of any array with a single SUBSCRIPT. An array with a single SUBSCRIPT is called a "list". This SORT works on lists of any length, and of any data type, including strings, integers, single precision numbers, and double precision numbers. The SORT is fast enough to invert a list of 1000 singLe precision numbers in about 5 seconds. It will order a randomly generated list of 3000 single precision numbers in about 2 minutes.

DIRECTION is used to determine whether the SORT is ascending or descending. This parameter may be any literal or variable string that begins with the letter "U" (for UP) or "D" (for DOWN). . Specifying "U" as the direction will cause an ascending SORT, and "D" will cause a descending SORT. The direction must be included in any SORT command that uses the index option, but it may otherwise be omitted, in which case the default SORT will be ascending.

The INDEX parameter is optional. When it is used, the SORT command requires that it be the zero index of a list that is the same size as the list specified by ARRAY. The INDEX list must be a list of integers. This list is supposed to contain the original location of each entry in the ARRAY. If ARRAY consists of names drawn from a disk file, then INDEX would consist of the record numbers from that file, so that the name in ARRAY(N) came from record number INDEX(N). When the INDEX option is used, the INDEX array will stay in correspondence with ARRAY. If INDEX is used, and the INDEX array is not a list of integers, an error will be generated.

Here are the vaLid SORT commands:

```
1.  CMD SORT,AR(0)
2.  CMD SORT,AR(0),"UP"
3.  CMD SORT,AR(0),"DOWN"
4.  CMD SORT,AR(0),"UP",NDX(0)
5.  CMD SORT,AR(0),"DOWN",NDX(0)
```

Note that all array names specify the zero index.

AN EXAMPLE OF INDEXING

In order to see what the indexing option does, try the following program:

```
10 CLS: CLEAR 50: DEFINT A-Z: DM = 12
20 DIM CURRENT(DM),OLD(DM),INDEX(DM)
30 FOR I=0 TO DM:
        CURRENT(I)=RND(10000):
        OLD(I)=CURRENT(I):
        INDEX(I)=I:
   NEXT
40 CMD SORT,CURRENT(0),"DOWN",INDEX
50 PRINT "POSITION      VALUE     PREVIOUS POSITION"
60 FM$ = "#####        #####     #####"
70 FOR I=0 TO DM:
        PRINT USING FM$; I, CURRENT(I),INDEX(I):
   NEXT
```

This program creates a random list of integers that are duplicated in an old list. The index of each element is stored in an index array. Notice that line 10 defines all variables as integers, so that the index array is an integer array. When the array is sorted with the index option on the index array, it returns with the data in the current array in order, and the data in the index array pointing to the original location of the current data. This can be seen by running this line from the command mode after the program has finished:

```
FOR I=0 TO DM: PRINT OLD(INDEX(I));: NEXT
```

The results of this line will be in the same order as the center column of the program output.

A NIFTY LITTLE DITTY

This SORT is implemented in such a fashion that when sorting alphanumeric lists (such as a list of names), the SORT will place all zero length strings at the tail of the list, whether the SORT is ascending or descending. This provides the user a simple test to determine when significant data ends. The first occurrence of a zero length string in a list will mean that all elements of the list beyond that point are also zero length strings.

FILENAME REDIM   INSTALL REDIM   REMOVE REDIM

COMMAND FORMATS

```
CMD REDIM ,  ARRAY LIST
CMD LOOKUP , SUBLIST , DATA , INDEX
```

THE LOOKUP COMMAND

The LOOKUP command is used to find a match to specified data within an array. This command will search the SUBLIST specified by the first parameter for the DATA contained within the second parameter. If the DATA is found, then the index of that DATA within the SUBLIST will be returned within the variable that is used as the INDEX parameter. If the DATA is not found, a -1 will be returned in the INDEX variable.

The "SUBLIST" can be any SUBLIST of an existing array. see the examples below to find out how a SUBLIST is specified.

The "DATA" can be any EXPRESSION, variable, or literal string. This parameter defines the DATA that the command is trying to look up.

The "INDEX" can be any single numeric variable. The results of the LOOKUP command are stored in this variable for later access by the user.

ONE QUICK NOTE

The following examples of the LOOKUP command all sound as though the zero index does not exist. This was done for the sake of simplicity. Zero indices are fully supported.

LOOKUP EXAMPLE 1

The simplest use for the LOOKUP command can be illustrated by assuming a list of 100 names that is held in an array that was dimensioned as NM$(100). If the operator has entered the name "JOHN JONES" into the variable A$, the LOOKUP command can be used to quickly determine whether or not the name "JOHN JONES" is in the list by issuing the command:

```
CMD LOOKUP, NM$(#), A$, INDEX
```

On return, the numeric variable "INDEX" will contain a -1 if the name does not appear in the list. If the name is in the list, then the INDEX variable will contain the position of the name within the list. A literal string could be used instead:

```
CMD LOOKUP, NM$(#), "JOHN JONES", INDEX
```

POINT NUMBER 1

Notice that the MARKER used by the LOOKUP command is the "#" symbol.

LOOKUP EXAMPLE 2

Now suppose that the list is made up of 100 numbers instead of 100 names, and that the list was dimensioned as SCORE(100). To make the example a little more solid, let's suppose that the numbers are test scores. If the operator inputs the number 78 into the variable XX, the LOOKUP command can be used to find out which (if any) of the test scores are equal to this number by issuing the command:

```
     CMD LOOKUP, SCORE(#), XX, INDEX
```

The position of the first entry in the list equal to 78 will return in the variable "INDEX". If there is no entry equal to 78, then "INDEX" will return as -1.

POINT # 2

When searching an array for a match between two numeric items, the LOOKUP command creates the "STR$" of each item before doing any comparison. This makes it possible to match NUMERIC items in spite of limitations due to questions of precision.

LOOKUP EXAMPLE 3

The really interesting applications for this command become apparent when we begin to consider arrays that are more than a simple list. In example 2, the LOOKUP command was used to find the INDEX of a test score within a list of 100 test scores. Now let's suppose that the test scores are stored in a array that was dimensioned to keep the scores on all individuals in 3 different classes, but let's assume that there are never more than 50 people in a single class. An array dimensioned as SCORE(50,3) is perfect for this purpose. The first SUBSCRIPT will pick out the individual, and the second SUBSCRIPT identifies the class.

Now suppose that this array has some entries, and we want to find out which person in class #2 got a test score of 78. Issue the command:

```
     CMD LOOKUP, SCORE(#,2), 78, INDEX
```

"INDEX" will return with the INDEX of the first person in class #2 that got a score of 78.

A NICE TOUCH

To continue this search from the point where it returned a match, issue this command:

```
     CMD LOOKUP , CONT , INDEX
```

This format of the LOOKUP command makes it possible to continue a search using previously established parameters. If the variable "INDEX" contains the value -1, then no further matches are possible, and trying to CONT will generate an error. This aspect of the LOOKUP command makes it possible to find all matches within any SUBLIST of any array.

LOOKUP'S FINAL EXAMPLE

Example three looked up which person in class #2 got a test score of 78. We could have used:

```
     CMD LOOKUP, SCORE(13,#), 78, INDEX
```

This command will return the class number in which the 13th person had a test score of 78. As a question, this seems pretty silly. The point is, the LOOKUP command does not care which SUBSCRIPT of an array is used to define a SUBLIST. Nor does it care how many

SUBSCRIPTs an array uses, as long as some SUBLIST is properly defined before the LOOKUP command is used.

IN CASE YOU WERE CURIOUS

You might wonder why the INDEX parameter is used in this command. It would be a lot easier to make an assignment to the INDEX variable rather than just hanging it on the end of the command. The problem is that the basic interpreter does not recognize the CMD token as a valid part of any kind of EXPRESSION. Therefore this command cannot be used on either side of the "=" sign. Another good question to ask about the LOOKUP command is this:

IF AN ARRAY IS NAMED BY DIM AR(1,2,3)
HOW MANY SUBLISTS ARE IN THE ARRAY ???

The answer is, it contains:

3*4 + 2*4 + 2*3 = 26 sublists.

Even an array dimensioned as ARRAY(1,1) will contain 4 sublists.

REDIM DISCUSSION

The REDIM command is used to redimension existing arrays. It may also be used to dimension arrays that do not exist. The format for the ARRAY LIST parameter is identical to the format used by the BASIC keyword DIM. The only difference is that any array name can be used in the REDIM command, whether or not it already exists. Any information contained in an array that is used in the ARRAY LIST will be lost.