FAST BASIC-V ROM

AUGUST, 1982
Copyright Infotek Systems

INFOTEK SYSTEMS, INC.
1400 North Baxter Street
Anaheim, California 92806

TABLE OF CONTENTS

FAST BASIC V

INTRODUCTION

The Infotek Systems Fast Basic V ROM is a continuation of software enhancements for the HP9830 A/B user. The 17 statements in FB-V offer additional speed and flexibility by building on existing ROM's in new, more powerful ways. The String Variables, Advanced Programming II, Infotek Systems Mass Memory II ROM, and ROM Clock, are all enhanced in performance through the use of FB-V statements.

The enhancements for the 9880 Mass Memory user are extremely versatile and powerful. All manner of disc operations previously impossible or very slow become practical and fast.

AP-II users gain an easy way to display and set flags with two new statements. Also, users of the Infotek Systems ROM clock have additional statements. The most powerful provides a timed interrupt which generates a GOTO for a service subroutine, an invaluable tool in real-time applications. Two additional statements aid in conversions between milliseconds, days, hours, minutes and seconds.

String Variable ROM users benefit from a simple but powerful statement which permits interactive string editing from the keyboard during a running program.

The power of the Infotek Systems MX-30 memory expander is further enhanced by two statements which allow branching to a service subroutine upon depression of a selected key. Also included is a statement which, under program control, determines if the machine is an MX-30 or not, simplifying commonality of software.

There is also a simple, yet elegant statement which allows redirection of default output for select code 15 to another select code, a must for applications using a CRT terminal in conjuction with the 9830.

As can be seen from the power of these new capabilities, Infotek Systems is strongly committed to making your 9830 a more powerful tool to solve new, more complicated problems.

EDIT Statement

The EDIT statement allows the user to alter a string variable using the 9830 display editing capabilities. EDIT displays the referenced string or substring and stores the altered version when Execute is pressed.

This statement can only be used in program mode.

EDIT requires the string ROM.

SYNTAX:

EDIT <STRING NAME> [<SUBSTRING DELIMITERS>]

EXAMPLES:

```
10 EDIT A$
```

```
10 EDIT A$(2,20)
```

```
10 INPUT A$(1,20)
15 REM CHECK IF INPUT STRING IS CORRECT
20 DISP A$(1,20,);"OK";
30 INPUT T
40 IF T = 1 THEN 70
45 REM EDIT STRING IF WRONG
50 EDIT A$(1,20)
60 GOTO 20
70 ...
```

ECHO Statement

The ECHO Statement re-directs all output for select code 15 to a different select code specified by the user. This includes output from TLIST, PRINT, and PRINTALL statements as well as formatted and unformatted writes to select code 15. Specifying a select code of 0 resets the default back to 15.

SYNTAX:

ECHO <SELECT CODE>

EXAMPLE:

```
10 ECHO 2
20 PRT-ALL 1
30 DISP "THIS IS NOW ECHOED ON SELECT CODE 2 INSTEAD OF 15"
40 ECHO 0
50 DISP "THIS ECHOED ON SELECT CODE 15 AS USUAL"
60 ...
```

DFLAG Statement

DFLAG displays all 16 AP II flags on the specified select code. 1 is displayed if the flag is set, and 0 otherwise. Flag 0 is displayed first and flag 15 last (rightmost).

If no select code is specified, then the flags are shown on the display.

SYNTAX:

```
DFLAG [# <SELECT CODE> ]
```

EXAMPLE:

```
10 CODE 16
20 SFLAG 1
30 SFLAG 12
40 DFLAG #15
```

on select code 15:

```
0100000000001000
```

TFLAG function

The TFLAG function returns the flag word of AP II, containing all 16 flags. This makes saving the flags very simple. In addition, if the argument is non-zero, then the flag word is set to the value given. Flag 0 is the low order bit in the word, flag 15 the high order.

EXAMPLE:

```
10 CODE 16
20 SFLAG 2
30 DFLAG #15
40 PRINT TFLAG(3)
50 DFLAG  15
```

when run, this would give the following output:

```
0010000000000000
4
1100000000000000
```

RPLAT Statement

The RPLAT statement transfers data directly from the 9880 disk to a specified array, starting at a specified element. One record at a time is transferred until either the next record would overflow the bounds of the array or the optional record count is exceeded. Any precision array is allowed, and data is read in word by word with no type conversions performed. Thus one record will fit in 256 integer, 128 split, or 64 full precision array elements.

RPLAT works on the unit currently selected.

RPLAT requires the Mass Memory II ROM.

SYNTAX:

```
RPLAT <HEAD >,<TRACK >,<SECTOR >,<ARRAY ELEMENT>
[,<RECORD COUNT>]
```

0 <= HEAD <=1 0<= TRACK <= 202 , 0<= SECTOR <=22

EXAMPLE:

if
```
10 DIM AI[24,256]
```
then:

```
RPLAT 0,0,0,A[1,1]
```
will read the entire main directory into A.

```
RPLAT 1,201,0,A[1,1],12
```
will read the first half of the spare directory into the first half of A.

WPLAT Statement

WPLAT is the converse of RPLAT, with data being written to the disk from an array. The transfer is terminated by the writing of the last array element, by exceeding the optional record count, or by reaching end of disk.

WPLAT works on the unit currently selected.

WPLAT requires the Mass Memory II ROM

SYNTAX:

WPLAT <HEAD>,<TRACK>,<SECTOR>,<ARRAY ELEMENT> [,<RECORD COUNT>]

EXAMPLE:

```
10 DIM AI[24,256]
20 RPLAT 1,201,0,A[1,1]
30 WPLAT 0,0,0,A[1,1]
```
this copies the spare directory to the main directory.

FPRINT# Statement

The FPRINT# Statement provides for fast array transfer to the 9880 disk. This is accomplished by not performing all the type checking and formatting performed by the MATPRINT# statement. As a result, the array is written to disk typically three times faster. FPRINT# can be used anywhere that MATPRINT# is used. The syntax of the two statements is the same.

Normal PRINT# and MATPRINT# statements can be interleaved with FPRINT# statements in the same data file. However, since FPRINT# writes out the array in a different format than PRINT# and MATPRINT#, an array written with FPRINT# can only be read using FREAD#. Attempting to use the Mass Memory ROM's READ# or MATREAD# will result in unpredictable results. Care must also be taken when performing random access in a file with arrays which have been written with FPRINT#.

Unlike MATPRINT#, integer arrays use only their length in words, not twice that value. This can result in disk space savings. The space used by an array with FPRINT# is the length of the array in words + 2. Thus if A is 4 x 30 integer array, it will use 122 words, as opposed to 240 with MATPRINT#.

FPRINT# requires the Mass Memory II ROM.

SYNTAX:

FPRINT# <FILE NO.> [, <RECORD NO.>]; <ARRAY NAME> [..., <ARRAY NAME>]

EXAMPLE:

```
FPRINT# 1,1;A,B
```

FREAD# Statement

FREAD# is the corresponding statement to FPRINT# to read an array from the disk. The array on the disk must be an array written with FPRINT#, or error 98 will result. In addition, the precision and length of the destination array must match the precision and length of the array that was written by FPRINT#, or error 98 will result. FREAD# must replace MATREAD# anywhere that the corresponding MATPRINT# statement was replaced by FPRINT#. FREAD# does not allow redimensioning as MATREAD# does.

FREAD# requires the Mass Memory II ROM.

SYNTAX:

```
FREAD# <FILE NO.> [,<RECORD NO.>]; <ARRAY NAME>
[...,<ARRAY NAME>]
```

EXAMPLE:

```
FREAD#1;A
FREAD#2,3;B,C
```

FTYPE Function

The FTYPE function may be used to determine whether the next element in the file specified was written by FPRINT#. If not, zero is returned. If so, then the return variable specifies the precision of the array written:

```
FULL = 7
SPLIT = 8
INTEGER = 9
```

SYNTAX:

```
FTYPE (<FILE NO.>)
```

EXAMPLE:

```
10 DIM AS [4,5]
15 ...
20 PRINT#1;"THIS IS A HEADER"
30 FPRINT#1;A
40 ...
100 READ#1,1
110 PRINT FTYPE(1)
120 READ#1;R
```

```
130 PRINT FTYPE(1)
```

would print:

```
0
8
```

FLEN Function

FLEN returns the length in words of the next array in the record if it was an array written by FPRINT#. If it was not written by FPRINT#, then zero is returned.

SYNTAX:

```
FLEN(<FILE NO.>)
```

EXAMPLE:

In the example for FTYPE, if the calls to FTYPE are replaced by calls to FLEN then the following is printed:

```
0
40
```

MSTDY Statement

The MSTDY statment converts a millisecond value into days, hours minutes, seconds and milliseconds.

This statement requires the ROM CLOCK.

SYNTAX:

```
MSTDY <MILLISECOND VALUE>,<DAY>,
<HOUR>,<MINUTE>,<SEC.>,<MILLISEC.>
```

EXAMPLE:

```
MSTDY 1E9,D,H,M,S,T
SETS:
D=11
H=13
M=46
S=40
T=0
```

Thus 1 billion milliseconds is 11 days, 13 hours, 46 minutes and 40 seconds.

DYTMS Statement

DYTMS converts days, hours, minutes and seconds into milliseconds. In conjunction with the MSTDY statement, this provides easy calculation of time intervals.

This statement requires the ROM CLOCK.

SYNTAX:

```
DYTMS <DAYS>,<HOURS>,<MINUTES>,<SECONDS>,<RETURN VAR>
```

EXAMPLE:

```
DYTMS 11,13,46,40,T
SETS T= 1E9
```

This is the inverse of the example for MSTDY.

ONKEY Statment

The ONKEY statement provides the user with a software controllable interrupt. When a key is struck while the calculator is running a program (and not in an INPUT statement) the running program will be interrupted. This means that at the end of the line currently being executed, control will be transferred to the line specified in the ONKEY statement. This is done just as if a GOSUB followed the currently executing line, so a RETURN statement will resume execution where it was interrupted.

After an interrupt, the key interrupt is disabled. Thus any further keys struck will not generate an interrupt unless another ONKEY statement is executed. The ONKEY statement should be the last statement before the return statement in the interrupt service routine.

The ONKEY statement will only operate on an MX-30-equipped 9830. Attempting to use it on a non MX machine will result in error 2000.

SYNTAX:

```
ONKEY <LINE NO.>
```

EXAMPLE:

```
10 ONKEY 100
20 DISP "NO KEY STRUCK"
30 GOTO 20
```

```
40 ...
100 DISP "KEY=" KEY (-1)
110 WAIT 1000
120 ONKEY 100
130 RETURN
```

The display will shown "NO KEY STRUCK" until the user strikes a key. Then an implicit GOSUB to line 100 will occur, where the value of the key struck will be displayed. Line 120 re-enables the key interrupt, and line 130 returns to line 20 or 30, depending upon where the interrupt occured.

```
10 ONKEY 1000
20 FOR I=1 TO 1E6
30 J=J+I/J
40 .....
90 NEXT I
1000 DISP"I="I
1010 WAIT 1000
1020 ONKEY 1000
1030 RETURN
```

This program will run normally until a key is struck (in a non-input state). Then the current value of the loop counter will be displayed. Thus a check can be made on the progress of involved calculations.

OFFKEY Statement

The OFFKEY statement disables the ONKEY interrupt if it is enabled. If not, there is no effect.

OFFKEY will only run on an MX-30. Attempting to execute OFFKEY on a non MX-30 will result in error 2000.

SYNTAX:

OFFKEY

CLKINT Statement

The CLKINT statement sets up the calculator for a pseudo-interrupt at the given time. This is accomplished by checking the value of the clock at the end of every statement and seeing if the time has arrived. If not, execution continues normally. If it has, a GOSUB is performed to the line given in the CLKINT statement.

The time given is a millisecond value corresponding to that returned by the MSEC function of the ROM CLOCK. A value of zero will disable the interrupt. The interrupt is also disabled if an interrupt occurs.

This statement requires the ROM CLOCK.

This statement will only run on an MX-30, otherwise error 2000 results.

SYNTAX:

```
CLKINT <MILLISECOND TIME VALUE>,<LINE NO.>
```

EXAMPLE:

```
10 CLKINT MSEC(0)+1E4,1000
20 FOR I=1 TO 1E5
30 ....
70 NEXT I
80 CLKINT 0,1
90 ...
1000 DISP"NOT FINISHED AFTER 10 SECONDS, CONTINUE"
1010 INPUT T
1020 IF T=0 THEN 1050
1030 CLKINT MSEC(0)+1E4,1000
1040 RETURN
1050 END
```

Line 10 enables an interrupt at 10 seconds (10,000 MS) from the current time. Line 80 disables the interrupt if the calculation finishes before the allotted time is up. If not, control is transferred to line 1000 and the user is asked if he wants the calculations to continue. If so, line 1030 re-enables the interrupt for 10 seconds later, and returns to continue the calculation. If not, the program terminates. Thus if it appears that the calculation is taking too long, the user may stop it.

BINEX STATEMENT

The BINEX statement allows the user to program any statement or command to be executed automatically after loading a binary program. This is extremely useful in linking compiled program segments into memory without having to manually start the subsequent program(s).

SYNTAX:

BINEX <Command/statement to be executed after LOADBIN*>

*Must be an even number of characters immediately following the last character of BINEX. A space counts as a character when using a command with an odd number of characters.

EXAMPLE:

The following demonstrates how to automatically link two BASIC language programs, both using a different binary program.

File organization on cassette or floppy

0	Start Program
1	First Binary Program
2	First BASIC Program
3	Second Binary Program
4	Second BASIC Program

The "start" program will load the first binary program in file 1 (line 30). The BINEX statement will load and run the first BASIC program in file 2.

Start Program, file 0:

```
10  REM FILE 0
20  BINEX LOAD #5,2,10,10
30  LOADBIN #5,1
40  END
```

A routine at the end of the BASIC program on file 2 can use the BINEX statement in a similar manner to load and run the binary program on file 3 and the BASIC program on file 4.

```
10  REM FILE 2
20
30
40
.
.
.
.
.
1000  BINEX LOAD #5,4,10,10
1010  LOADBIN #5,3
1020  END
```

BINEX (execute) or
10 BINEX (end of line)
cancels any previous BINEX command. File 4, for example, can contain a BINEX statement to cancel the command in file 2.

Note: The MM II statement "GETB," can be used in the same manner as LOADBIN with the BINEX statement.

<u>ISMX FUNCTION</u>

ISMX returns 1 if the calculator is equipped with an MX-30; otherwise 0 is returned. It requires a single dummy argument. The dummy argument variable must first be defined.

SYNTAX SUMMARY

```
   BINEX <statement or command to be executed after LOADBIN OR GETB,>
  *CLKINT <MISSLISECOND TIME>,<LINE NO.>
  +DFLAG [# <SELECT CODE>]
  *DYTMS <DAY>,<HOUR>,<MINUTE>,<SECOND>,<MILLISECOND RETURN VAR>
***EDIT <STRING NAME> [<SUBSTRING DELIMITERS>]
   ECHO <SELECT CODE>
  -FREAD# <FILE NO.> [,<RECORD NO.>] ; <ARRAY NAME>
   [...,<ARRAY NAME>]
  -FLEN(<FILE NO.>)
  -FPRINT# <FILE NO.> [,<RECORD NO.>] ; <ARRAY NAME> [<ARRAY NAME>]
  -FTYPE(<FILE NO.>)
   ISMX(<DUMMY>)
  *MSTDY <MILLISECOND>
   VALUE>,<DAY>,<HOUR>,<MINUTE>,<SECOND>,<MILLISECOND>
 **OFFKEY
 **ONKEY <LINE NO.>
  -RPLAT <HEAD>,<TRACT>,<SECTOR>,<ARRAY ELEMENT> [,<RECORD COUNT>]
  +TFLAG( X )
  -WPLAT <HEAD>,<TRACK>,<SECTOR>,<ARRAY ELEMENT> [,<RECORD COUNT>]

   *   = ROM Clock
   -   = MM II
   +   = AP II
   **  = MX-30
   *** = String ROM
```