MMBasic V3.X Features

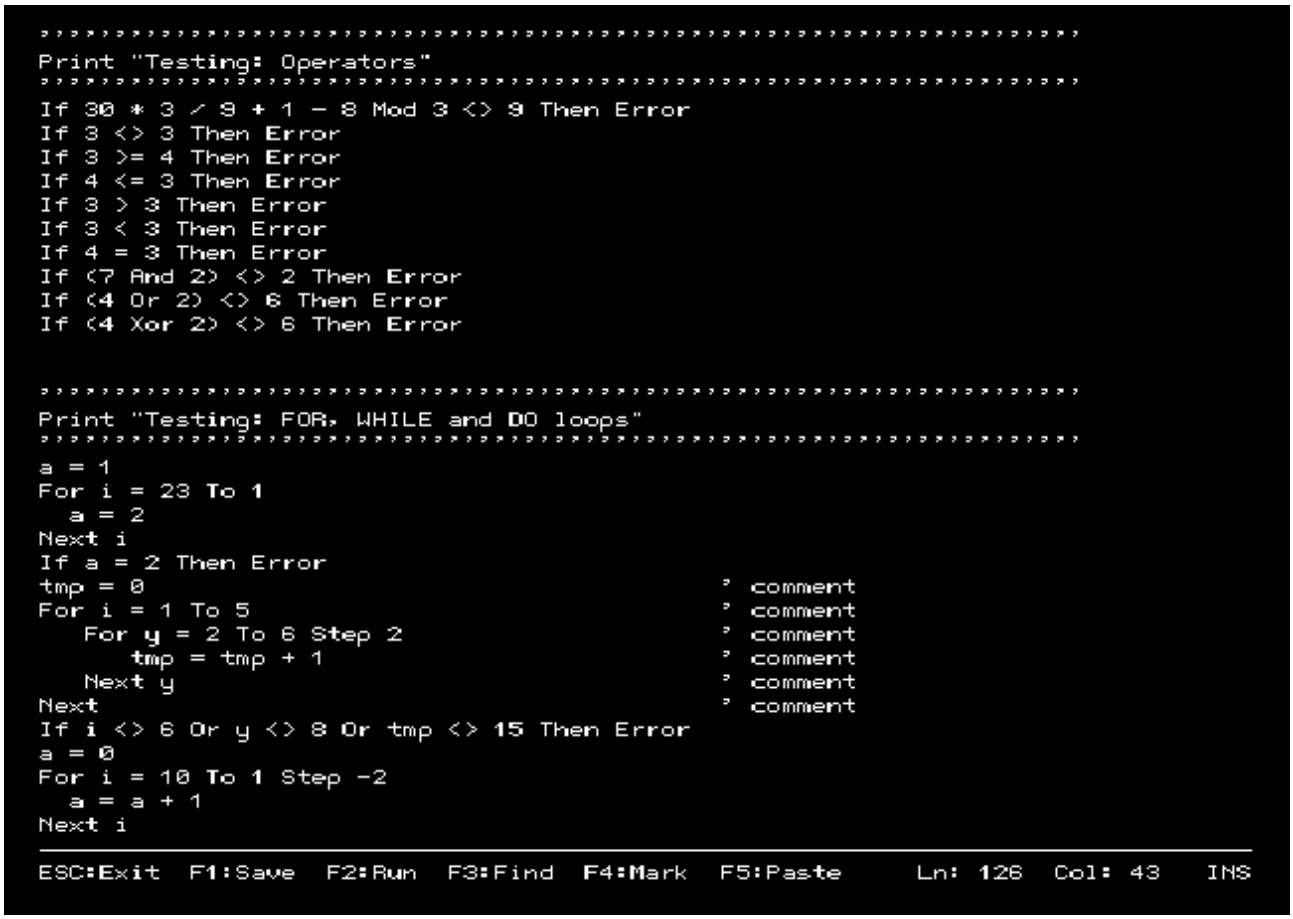
Version 3.0 introduced a revolutionary change in MMBasic compared to version 2.7B. In addition to dozens of small improvements and bug fixes version 3.0 also implemented an updated core of the BASIC interpreter which provides the basis for many significant changes that would have previously been impossible to implement.

Originally MMBasic was written to emulate the early BASICs like Microsoft's MBASIC. This was not a problem if you were over 50 and grew up in the days of the Tandy TRS-80 and Commodore 64 as you were used to the languages of that era and their idiosyncrasies. But, with the huge popularity of MMBasic, it needed to move beyond the days of the TRS-80 to match modern programming standards.

This has been done with a rewrite of the interpreter core that enables features such as speed improvements, the removal of line numbers and the addition of labels, user defined subroutines/functions and more.

Full Screen Editor

An important productivity feature of 3.X is the full screen editor (this is not available in the DOS version of MMBasic).



The editor is invoked with the EDIT command. If you run it on its own it will automatically start editing the program currently in memory. If you run it with a file name (eg, EDIT "FILE.DAT") it will edit that file while leaving the program in memory untouched. This last feature is very handy for examining data files or editing font files while you are developing a program.

When you are in the editor the various editing keys will work the way that you would expect. For example, the Up Arrow key will move up a line, the Delete key will delete a character and so on. You can quickly move through the text from the beginning to the end using the Home/End and Page Up/Down keys.

Using the full screen editor makes programming in MMBasic a much more fun experience. You can run the editor, make your changes and press the F2 key. The program will be saved back to program memory and automatically run. If it fails with an error you can run the editor again and the editor will automatically position the cursor at the line that caused the error.

The editor will also work with a vt100 compatible terminal emulator over USB. So, if you are using the mini Maximite or the UBW32 you can still conveniently edit the program held in memory. It has been tested with Tera Term and this is the recommended terminal emulation software. Note that Tera Term must be configured for an 80x36 sized screen.

Line Numbers

Another significant change is that line numbers are now no longer required in your BASIC programs.

For example:

MMBasic 2.X

```
                ...
324 ' GET THE INPUTS
330 IF Q=0 THEN GOTO 700
331 A=A+Q : S=S-Y*Q : C=0
334 GOTO 400
336 ' GET THE SELL STATUS
340 PRINT "DO YOU WISH TO SELL";
341 INPUT Q
345 IF Q<0 THEN GOTO 340
350 A=A-Q : S=S+Y*Q : C=0
400 PRINT
410 PRINT "HOW MANY";
411 INPUT Q
420 IF Q<=S THEN GOTO 324
430 S=S-Q : C=1 : PRINT
                ...
```

MMBasic 3.X

```
                ...
      ' Get the inputs
GetI: If Q=0 Then GoTo Abort
      A=A+Q : S=S-Y*Q : C=0
      GoTo GetQ

      ' Get the sell status
GetQ: Print "DO YOU WISH TO SELL";
      Input Q
      IF Q<0 Then GoTo GetQ
      A=A-Q : S=S+Y*Q : C=0
      Print
      Print "HOW MANY";
      Input Q
      IF Q<=S Then GoTo GetI
      S=S-Q : C=1 : Print
                ...
```

Removing line numbers makes it much easier to write clear and meaningful programs. You can indent lines for clarity and the ability to insert blank lines allows you space out the program so that it is not one solid block of text.

Version 3.X is fully backward compatible with 2.X so you can still use line numbers if you need to and old programs written for 2.X will run exactly as before.

Labels

Instead of using line numbers as the target for GOTO, GOSUB, etc in version 3.X you can use a label.

In the above example GetQ is a label. A label is used exactly like a line number but it is much more useful to anyone who is reading the program. For example:

```
GOTO ErrorHandler
```

Is more meaningful than:

```
GOTO 1290
```

When used to mark a line the label must be terminated with the colon (:) character. For example:

```
ErrorHandler:  Print "Error …
```

You can also mix labels and line numbers.

For example:

```
100 ' old style program that uses labels
200 Label: Print nbr
400 nbr = nbr + 1
500 if nbr < 5 GoTo Label
```

A label has the same specifications as a variable (up to 32 characters including letters, numbers, the underscore and period). The only caveat is that a label cannot be the same as a command because that could confuse the interpreter as the colon character (:) is used to both terminate a label and as a separator between commands.

MMBasic includes a cache which remembers the location of a label in the program and, as a result, jumping to a label is much faster than jumping to a line number. For that reason labels are preferred and should be used instead of line numbers.

Defined Subroutines

Defined subroutines are a useful feature to help in organising programs so that they are easy to modify and read.

A defined subroutine is simply a block of programming code that is contained within a module and can be called from anywhere within your program. It is the same as if you have added your own command to the language.

For example, assume that you would like to have the command FLASH added to MMBasic, its job would be to flash the power light on the Maximite. You could ask the author of the language to add it or you could define a subroutine like this:

```
Sub FLASH
  Pin(0) = 1
  Pause 100
  Pin(0) = 0
End Sub
```

Then, in your program you just use the command FLASH to flash the power LED. For example:

```
IF A <= B THEN FLASH
```

If the FLASH subroutine was in program memory you could even try it out at the command prompt, just like any command in MMBasic.

The definition of the FLASH subroutine can be anywhere in the program but typically it is at the start or end. If MMBasic runs into the definition while running your program it will simply skip over it.

Subroutine Arguments

Defined subroutines can have arguments (sometimes called parameter lists). In the definition of the subroutine they look like this:

```
Sub MYSUB (arg1, arg2$, arg3)
  <statements>
  <statements>
End Sub
```

And when you call the subroutine you can assign values to the arguments. For example:

```
MYSUB  23, "Cat", 55
```

Inside the subroutine `arg1` will have the value `23`, `arg2$` the value of `"Cat"`, and so on. The arguments act like ordinary variables but they exist only within the subroutine and will vanish when the subroutine ends. You can have variables with the same name in the main program and they will be different from arguments defined for the subroutine (at the risk of making debugging harder).

When calling a subroutine you can supply less than the required number of values. For example:

```
MYSUB  23
```

In that case the missing values will be assumed to be either zero or an empty string. For example, in the above case `arg2$` will be set to `""` and `arg3` will be set to zero. This allows you to have optional values and, if the value is not supplied by the caller, you can take some special action.

You can also leave out a value in the middle of the list and the same will happen. For example:

```
MYSUB  23, , 55
```

Will result in `arg2$` being set to `""`.

Local Variables

Inside a subroutine you will need to use variables for various tasks. In portable code you do not want the name you chose for such a variable to clash with a variable of the same name in the main program.

To this end you can define a variable as LOCAL. For example, this is our FLASH subroutine but this time we have extended it to take an argument (nbr) that specifies how many times to flash the LED.

```
Sub FLASH ( nbr )
  Local count
  For count = 1 To nbr
    Pin(0) = 1
    Pause 100
    Pin(0) = 0
    Pause 150
  Next count
End Sub
```

The counting variable (`count`) is declared as local which means that (like the argument list) it only exists within the subroutine and will vanish when the subroutine exits. You can have a variable called `count` in your main program and it will be different from the variable `count` in your subroutine.

If you do <u>not</u> declare the variable as local it will be created within your program and be visible in your main program and subroutines, just like a normal variable.

You can define multiple items with the one LOCAL command. If an item is an array the LOCAL command will also dimension the array (ie, you do not need the DIM command). For example:

```
LOCAL NBR, STR$, ARR(10, 10)
```

You can also use local variables in the target for GOSUBs. For example:

```
  GOSUB MySub
       ...
MySub:
   LOCAL X, Y
   FOR X = 1 TO ...
   FOR Y = 5 TO ...
  <statements>
   RETURN
```

The variables X and Y will only be valid until the RETURN statement is reached and will be different from variables with the same name in the main body of the program.

Defined Functions

MMBasic version 3.2 introduced defined functions. These are similar to defined subroutines with the main difference being that the function is used to return a value in an expression.

For example, if you wanted a function to select the maximum of two values you could define:

```
Function Max(a, b)
  If a > b
    Max = a
  Else
    Max = b
  EndIf
End Function
```

Then you could use it in an expression:

```
SetPin 1, 1 : SetPin 2, 1
Print "The highest voltage is" Max(Pin(1), Pin(2))
```

The rules for the argument list in a function are similar to subroutines. The only difference is that brackets are required around the argument list when you are calling a function (they are optional when calling a subroutine).

To return a value from the function you assign a value to the function's name within the function. If the function's name is terminated with a $ the function will return a string, otherwise it will return a number. Within the function the function's name acts like a standard variable.

As another example, let us say that you need a function to format time in the AM/PM format:

```
Function MyTime$(hours, minutes)
  Local h
  h = hours
  If hours > 12 Then h = h - 12
  MyTime$ = Str$(h) + ":" + Str$(minutes)
  If hours <= 12 Then
    MyTime$ = MyTime$ + "AM"
  Else
    MyTime$ = MyTime$ + "PM"
  EndIf
End Function
```

As you can see, the function name is used as an ordinary local variable inside the subroutine. It is only when the function returns that the value assigned to `MyTime$` is made available to the expression that called it. This example also illustrates that you can use local variables within functions just like subroutines.

Passing Arguments by Reference

If you use an ordinary variable (ie, not an expression) as the value when calling a subroutine or a function, the argument within the subroutine/function will point back to the variable used in the call and any changes to the argument in your routine will also be made to the supplied variable. This is called passing arguments by reference.

For example, you might define a subroutine to swap two values, as follows:

```
Sub Swap a, b
  Local t
  t = a
  a = b
  b = t
End Sub
```

In your calling program you would use variables for both arguments:

```
Swap nbr1, nbr2
```

And the result will be that the values of `nbr1` and `nbr2` will be swapped.

Unless you need to return a value via the argument you should not use an argument as a general purpose variable inside a subroutine or function. This is because another user of your routine may unwittingly use a variable in their call and that variable will be "magically" changed by your routine. It is much safer to assign the argument to a local variable and manipulate that instead.

Additional Notes

There can be only one END SUB or END FUNCTION for each definition of a subroutine or function. To exit early from a subroutine (ie, before the END SUB command has been reached) you can use the EXIT SUB command. This has the same effect as if the program reached the END SUB statement. Similarly you can use EXIT FUNCTION to exit early from a function.

You cannot use arrays in a subroutine or function's argument list however the caller can use them. For example, this is a valid way of calling the `Swap` subroutine (discussed above):

```
Swap dat(i), dat(I + 1)
```

This type of construct is often used in sorting arrays.

The use of defined subroutines and functions should reduce the need to add specialised features to MMBasic. For instance, there have been a few requests to add bit shifting functions to the language. Now you can do that yourself… this is the right shift function:

```
Function RShift(nbr, bits)
   If nbr < 0 or bits < 0 THEN ERROR "Invalid argument"
   RShift = nbr\(2^bits)
End Function
```

You can now use this function as if it is a part of the language:

```
a = &b11101001
b = RShift(a, 3)
```

After running this fragment of code the variable `b` would have the binary value of `11101.`

The defined subroutine and function is intended to be a portable lump of code that you can insert into any program. This is why the full screen editor has the CTRL-F keys for inserting another file. The idea is that you can keep your defined routines in a file and whenever you need them you can quickly insert them using CTRL-F.

So, it would be easy to create a library of bit manipulation functions like that described above and insert them into any program when needed.

Dynamic Memory Management

The earlier versions of MMBasic for the Maximite family employed a fixed memory layout for the program, variables and general use. When you ran out of space in one area there was no way to use the free space in another area.

Version 3.2 introduced Dynamic Memory Management. This dynamically allocates the memory to individual areas as required with the allocation being made from one single pool of memory. This pool can be quite large. These are the current sizes:

109KB with the video turned off
100KB with composite video
83KB with VGA video

Compare this to earlier versions that restricted the amount of memory for programs to 30K.

You can see the difference when you run the MEMORY command. It now lists the various areas and their use of the general pool:

```
13kB (12%) Program (541 lines)
 4kB ( 3%) 33 Variables
 0kB ( 0%) General
91kB (85%) Free
```

While the overall amount of memory in the PIC32 remains the same this system makes more efficient use of it. This translates into the ability to run very large programs (greater than 3,000 lines) or to allocate huge arrays with more than 20,000 elements.