The M Programming Language

Written by Paul Robson 2018

The basic concept behind the M programming language came from issues with Z80 Forth. The basic problem with Z80 Forth is that the processor only has one stack. You can use it for the data stack, the return stack, or with a bit of ingenuity, both. But it doesn’t work that well.

So rather than engineering for the Z80 to produce a certain language, it was decided to fit the language to the Z80. This version is the first version completed, and is known as M6 (there is a logical reason for this).

What M does is to reduce the working stack to two 16 bit registers known as A and B (in practice the HL and DE registers). It’s stack becomes a simple call stack. Much of the syntax is begged or borrowed from FORTH and ColorFORTH.

This means that more variables are used as it isn’t practical to keep too many intermediate values on the stack, though it can make more use of it than you’d think. Additional functionality is provided in the compiler to support variables and arrays. It also has the benefit that you don’t have to worry about the stack too much, you can’t overflow it !

The FORTH arithmetic looks fairly similar “4 1 -” works the same as it always did, even though internally it operates differently.

|  |  |  |  |
| --- | --- | --- | --- |
| **Command** | **Notes** | **A** | **B** |
| 4 | Put old value of A (not known) in B, set A to 4 | 4 | ? |
| 1 | Put value of A (4) into B, set A to 1 | 1 | 4 |
| - | Subtract A from B, put result in A. Don’t change B | 3 | 4 |

There are two types of words in M6.

**Constants**

The first is a constant. This can either be decimal (-42) hexadecimal ($2A7) or string “hello\_world”. The underscore is a placeholder for space to keep the one word one function idea and simplify the parser.

All place a constant into the A register ; in the case of a string it places a pointer to an M string which consists of a byte length followed by string data, e.g. it is *not* ASCIIZ but like classic FORTH.

Constants are an example of a **loader**. This is an important concept in M. Loaders copy A to B before activating, so what constant “42” does is actually to (i) move the original value of A into B and (ii) put 42 into A. This concept gives it its ‘FORTH’ look.

Other loaders are the address and load modifiers (see later).

**Words**

Everything else is an executable word. This is like a subroutine threaded word in FORTH. There are however some variations on this.

Words can be marked *immediate.* This means they execute when they compile, rather than compiling CALL <some address> they can generate code in their own right.

Words can be marked *compileonly*. This is used for words that cannot be executed interactively. This is because words are ‘executed’ by compiling code to load A and B from registers, generating the compiled code, and writing them back. Some words, like those which manipulate the return stack, or are two part words (e.g. *IF .. THEN*) cannot work in this environment.

Words can be *macros*. This is a special subtype and one of the features, along with the abandonment of the data stack, gives speed. Words can generate inline code ; for example ‘+’ which adds B into A is a single instruction add hl,de. Rather than generating a call to a single instruction, it simply puts out add hl,de.

Inline macros are immediate words , but can still be executed interactively (with the above compileonly exceptions). The + example looks as follows :

Call ExpandMacro ; expand macro routine, gets address off stack

Db 1 ; number of bytes in the macro

Add hl,de ; the actual macro itself.

**Modifiers**

All words can have modifiers applied to them that affects their behaviour. This is used to provide a variable and array functionality which replaces the stack manipulation.

Variable words are two byte variables defined using the variable keyword (e.g. *variable count*). Arrays are defined using the array keyword (e.g. *array test 42*) Arrays can be any size. All variables and arrays are private, which means they cannot be used outside the current code file.

Using a word ‘count’.

count Calls the code at count (which will probably crash)

count& Loads the variable count’s address into A (a loader)

count@ Loads the variable count into A (a loader)

count! Saves A into the variable count

count# Doubles A and add to the count address

Some of these are loaders, which do A→B first.

Accessors to variables and arrays should be done by appropriate words. This is because the cross paging mechanism required for the Spectrum Next can cope with cross page calls but not data access.

Some of these make no real sense. You would not want to do a call to a variable ; this does not, as in FORTH, provide a default behaviour but calls the address of the variable ; you are unlikely to use count@ etc. if it is a code word.

For this reason words can be marked *variable.* This means that doing count where count is a variable will cause an error (as I can see no scenario where doing this makes sense). However, this is not compulsory in a compiler, it is advised, because of the confusion with FORTH ; it is easy to forget the & when you want to use the address ; you could write 12 count +! when you actually mean 12 count& +!

**Core Words**

There is a core word set which is written in Z80 assembler. This can be expanded as necessary. Porting M involves rewriting the word set for your target and amending the compiler.

**Compiler**

The python compiler is based around a base file, which has the suffix .m6 . This can contain double slash comments, and almost everything else is M6 words as defined above, with the following exceptions

*import <name>*

Imports the module <name>.m6 from the library search path. Modules can only be imported once.

Note that an error is generated if an imported module (or the original program) causes a page change. This is so variables in modules are guaranteed accessible.

*list on|off*

Turns m6’s very basic listing functionality off and on.

**Dictionary Format**

+0 Offset to next entry (or zero), length of the word name + 5

+1,2 Word Address

+3 Word Page Number

+4 Word information

Bit 7 : immediate word

Bit 6 : compile only word (cannot run interactively)

Bit 5 : private flag.

Bit 4 : optional variable flag

Bit 0..3 : unused (should be zero)

+5 First character of name 6 bit ASCII lower case

+6 Second character (etc.)

+n-1 Last character of name, has bit 7 set to signify this.

**Stack problems**

A FORTH programmer will have notice several issues with this design. One is, how do you calculate something like (a + b) \* (c + d) as you don’t have the stack to keep the intermediate result. You can use a variable, or better, the return stack as a temporary store.

a@ // Read a and b

b@

+ // a+b result is now in A

a>r // put on return stack

c@ // Read c and d

d@

+ // c+d result is now in A

r>a // put c+d into B (a loader) and restore a+b to A

\* // multiply them together

This is fairly straightforward once you realise that r>a is a loader and does the A→B transfer before loading ; you don’t have to manually put A into B

**Parameters**

Obviously only 2 parameters are available, A and B, so words like copy and fill which take three have to use a memory block to pass them. There are words that generate the code that preserves stack values ; supposing you wanted a routine that incremented a specific variable for example

: incr 1 count& +! ;

Unlike FORTH this is destructive. It destroys A + B which you may not want. The push and pop words work round this e.g.

: incr push.ab 1 count& +! pop.ab ;

There is also a pop.bb which allows you to return a value in A while preserving B.

**Language Conclusion**

It seems bonkers but it does actually work quite well, keeping much of the benefits while producing quick and fairly compact code – obviously it can’t really compete with FORTHs that use lists of 16 bit addresses but it’s much quicker.

An interesting side effect is that it’s much harder to lose things on the stack because …. there pretty much isn’t one.