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. M has existed in various forms but it always follows this basic idea. The name comes from a very old programming language M5 which was on the Nascom 1, which it doesn’t resemble at all, but was the genesis of the language.

What current 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). Its stack becomes a simple call stack. Much of the syntax is begged or borrowed from FORTH and ColorFORTH. However, it doesn’t exist on the Next (or any other Z80 platform) as standalone runtime ; you cross develop using the compiler and an emulator.

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. To help with this the variable syntax is extended to concatenate together the variable word and the ‘@’ and ‘!’ operators to store and load directly.

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 four main types of command, two of which are shown here. This might seem bizarre, but it does actually work. Stick with it :)

1. Loading commands

The first 2 (4 and 1) are loading commands, which create a new value in A unrelated to the previous one. These can be numeric or string constants, variable addresses or reads, or r>a which transfers a value from the return stack to A (much the same as FORTHs R> command). Certain other words ; e.g. read a key from the keyboard, also behave in this fashion.

2. Binary commands

These involve using the two values to produce a third. These operate as B <op> A for those operations where order matters (subtract and divide, non equality comparisons). They change A but not B. These are + - \* / mod and or xor = <> >= > <= < max and min (among others possibly)

3. Unary commands

These involve working on a value in A that’s already there, and they don’t change B either. They just change A. They include 0= 0< not 0- abs 2\* 2/ 1+ 1- 2+ 2- and also @ and c@ which are the same as FORTH, 16 and 8 bit reads.

4. The rest

Then there are the remainder, many of which use the A and B as two parameters, e.g. the store commands ! c! +! and -!. Mostly these don’t change A or B, though there are some specific manipulation commands such as swap (swaps A+B over) and a>b and b>a which can be used in copy loops to avoid the use of a variable.

**Word Classes**

There are 3 types of word class. Each of these can be marked private, which means they are not accessible outside the current source file.

1. Code Words

These are words built up out of other words, like most FORTH words in Z80 code. To activate them they are CALLed. Code Words can be marked immediate, which means when you compile them they are executed instead rather than CALL code to them being generated, again as in FORTH, these are used as structures in the same way.

2. Macros

Many words are very simple. For example ‘swap’ is ex de,hl and ‘+’ is add hl,de. There is no point in using a CALLable routine for these. Macros are bits of code that are copied when they are compiled, so rather than ‘+’ generating a Z80 call it simply copies the code add hl,de in. Thes are executed in the same way as code words, but are prefixed with ld a,<length> ; call ExpandMacro followed by the macro code. Some macros are longer because of their frequency of use, especially @ and -.

3. Variables and Arrays

Variables are handled slightly differently. To save code memory primarily they do not have attached code ; a variable is simply a reference to a 2 byte area of memory (or a larger one if it is an array or other data structure). The compiler handles the code production.

Fundamentally, if you have a variable ‘count’

count puts the address of count into A, putting the old A into B (loader)

count[] adds the variable base address to A without changing anything else

count@ puts the content of count into A, putting the old A into B (loader)

count! puts A into the count variable, changing nothing else.

There is a simple equivalence between the last two and ‘count @’ and ‘count !’ except they can be run using a single Z80 instruction and you lose the stacking and destacking of the address and data value.

**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.