**M8 Language Specification**

This document specifies the latest iteration of my quasi FORTH for the Z80, M8. The idea for M8 was drawn from the difficulties with Forth on a Z80, viz. that FORTH has two stacks, and a Z80 has one stack and a rather slow alternative.

So rather than trying to produce a stack-based language, M8 does away with the data stack. The main consequence of this is that interim values cannot be stored on the stack, so there is a greater use of variables. M8 also borrows some ideas from ColorForth.

The register model of M8 is three 16-bit registers, known as “A” “B” and “C” and a 16-bit return stack. In the Z80 version (no other version exists, but there’s no reason why not!), A is HL, B is DE, C is BC. C is very much a temporary register used for things where M8 is weak ; copying in loops, complex expressions. It can do these, C just makes it easier.

Language

M8 has two types of words; firstly constants which are numeric in either decimal or hexadecimal format (-402 $1CA7)

These put a constant in A (*having previously copied A to B*). Everything else is a word which is a reference to Z80 code.

Words can be marked macro; this means that they are run when compiling them (e.g. immediate) ; the M8 compiler normally generates code for a word as the Z80 Call instruction otherwise.

Often this is not very efficient, as the word can be represented in three or fewer bytes, or alternatively it is considered ‘worth’ the extra byte or two for the speed benefit. For example, “swap” (which swaps A and B) is just ex de, hl. So, the Macro version compiles in the code for ex de, hl.

Words can exist in the dictionary in a macro and non-macro form. This often is done by defining the word as “: @ @ ;” (so to speak) this defines a non-macro @ which has the code generated when calling the macro @.

This is because macros cannot be run interactively; when you are executing a word rather than compiling it only searches for non-macro words.

M8 borrows ColorForth’s syntax system; white for comments, green for compile, yellow for immediate execution and red for word definition.

Core Words (missing structures and some kernel words)

|  |  |  |  |
| --- | --- | --- | --- |
| **Word** | **Type** | **Where** | **Description** |
| - | both | atomic | Subtract A from B, result in A |
| \* | word | internal | Multiply A by B, result in A |
| swap | both | atomic | Exchange A and B |
| / | word | internal | Divide A into B, result in A (unpredictable if A = 0) |
| + | both | atomic | Add B into A |
| > = <  >= <> <= | word | words | Compare A against B, set A to -1 if true, 0 otherwise |
| 0- | word | atomic | Negate A (2’s complement) |
| 0< | word | atomic | Set A to -1 if A <0, 0 otherwise |
| 0= | word | atomic | Set A to -1 if A = 0, 0 otherwise |
| 1+ 1- 2+ 2- | both | atomic | Adjust A by 1 or 2 |
| 2\* 4\* 8\*  16\* 256\* | both | atomic | Shift A left |
| 2/ 4/ 16/ | mixed | atomic | Divide A by 2 and 4 and 16 (arithmetic not logical) |
| ; | macro | atomic | Return from subroutine |
| r1 > r2 | both | atomic | Copy r1 to r2 where rx is A B or C (r1 <> r2 !) |
| rrr > stack  stack > rrr | macro | atomic | Push and restore A, B, C or some combination to or from the return stack |
| abs | word | atomic | Set A to abs(A) |
| and or xor | word | words | Binary operator; A:= A <op> B |
| bswap | both | atomic | Byte swap A |
| buffer  buffer.size | word | internal | Puts the address or size of the buffer into A |
| c! ! | both | atomic | 8 and 16 bit writes of B to address A |
| c@ @ | both | atomic | 8 and 16 bit reads of address A into A |
| clrscreen | word | internal | Clear Screen |
| copy | word | internal | Copy C bytes from B to A (copes with overlap) |
| cursor! | word | internal | Set the cursor position to A |
| debug | word | atomic | Display A B C on the screen |
| fill | word | internal | Fill C bytes from A with constant B |
| halt | word | internal | Stop |
| key@ | word | internal | Scan keyboard for ASCII (7 bit) keys, colour keys, cursor keys, backspace,break or cr, 0 if none |
| max min | word | words | Set A to the larger or smaller of A and B |
| mod | word | internal | Divide A into B, modulus of result in A |
| not | word | atomic | Not A (1’s complement) |
| port@ port! | macro | atomic | Read and write Z80 ports |
| screen! | word | internal | Set the screen byte A to address B (2 + 6 bit) |
| sys.info | word | internal | Puts the address of the system info area into A |

Memory Usage

The Z80 memory usage is a continuous block of memory. There are two versions ; in one the system starts at $8000, and in the other it starts at $5B00. The reason for this is the first supports the Spectrum Next’s low-resolution screen (which uses $6000-$8000) and the second makes more efficient use of lower memory. The upper 16k of memory can be paged, code access, but not data access can be done across pages, which is why variables are normally marked *private* to a module.

Memory usage is as follows (in order) :

* Boot code
* Helper functions for words
* Kernel Compiler
* Built in words written in assembler
* Return Stack
* System Variables
* Dictionary
* Program

The core system consists of the compiler. Code is then written using the compiler by text in 2+6 format being placed in screen memory between $4100 and $59FF which can be used to ‘boot’ the system (The SNASM assembler uses memory near $4000 for its SNA boot)

Some words can be directly included in the helper code and given a label marker ( ; @word <name> [MACRO] ) this is converted into a suitable comment and a label made up from the ASCII lower case values in hex of the name (e.g. ace becomes definition\_61\_63\_65) ; a macro will have a \_MACRO suffix to indicate this.

The compiler scans through the text ; where there is a red define, it creates a definition, where there is a yellow executable word it executes the word (not if it is a macro) wrapped in code to maintain/update A and B (for a constant it is direct, remembering the ‘pre load switch’).

Finally if the word is a green compiling word, it either executes the macro (again maintaining/preserving state) or compiles a call to the code.

Text Colour Forth

Obviously Color Forth’s boot code cannot be coded in Color Forth – initially anyway. I haven’t found an editor that is cross platform and useable.

So it is written in monochrome text and converted as follows :

// <comment> Comments (ignored)

:<word> Red defining word

<word> Green compilation word or number

[ <word> ] Yellow executable words

< <word> > Compile code inline

Words can be either constants , words or z80 opcodes which compile to one or two bytes. Opcodes are in curly brackets.

The <x> syntax does [{x}] [a,] i.e. it is direct assembly.

Modifiers

When declaring a variable (*variable demo)* it actually creates three words (demo@ demo! demo&) that are marked as private macros , normally not allowed.

These load, save and get the address of that variable ( thus saving the space and cost of demo ! demo @ ).

The two that load values (demo@ and demo&) copy A to B beforehand as constants do ; so you can write

demo@ 1+ demo!

to increment variable demo.