**M Language Specification**

This document specifies the latest iteration of my quasi FORTH for the Z80, M. The idea for M 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, M 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. M also borrows some ideas from ColorForth.

The register model of M 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 IX. C is very much a temporary register used for things where M is weak ; copying in loops, complex expressions. It can do these, C just makes it easier.

**Language**

M has two types of words; firstly constants which are numeric in either decimal or hexadecimal format (-402 $1CA7), or a string surrounded by double quotes “hello\_world”. The underscore is a placeholder for space.

These put a constant in A (*having previously copied A to B*). For a string constant it puts a pointer to the string in A (with A being copied to B), the string is a length prefixed byte list in 6 bit ASCII.

Everything else is a word which is a reference to Z80 code.

Words can be macros ; this means that they generate code when “compiling” ; the M compiler will 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.

Variables are defined using the variable keyword. in practice a variable is three macros – var@ var! and var& which read, write and get address of the variable. Var doesn’t work on its own.

Words and variables can be marked private, which means at the end of each module they are purged from the dictionary.

**Memory Usage**

M is a cross compiler. M code is fit into a continuous block of memory which can include paged memory.

This is implemented using the bootloader. The bootloader is an .SNA file (a standard Spectrum 48k file) that loads a file “boot.img” into the memory of the Spectrum Next.

This is loaded to $8000-$BFFF (the first 16k) and the remainder is loaded into the 64 x 8k pages numbered 32..95 (on the 8k mapping). Each of these is paired to be mapped into $C000-$FFFF at the same time, so the mapping looks a bit like a souped up ZX Spectrum 128, with memory from $4000-$BFFF static and 16k mapped in and out of $C000-$FFFF.

Memory from $4000-$7FFF is not used by default. The reason for the abandonment of $5B00-$7FFF is its use in Low Resolution mode. If the developer doesn’t want to use this mode, they can do what they like with it.

The boot.img contains 64 x 8k pages and 1 x 16k block which means it occupies 540,672 bytes of storage.

The dictionary exists in a file “boot.dict” which has basic dictionary information ; routine address and page, type (macro or word).

These can be repeatedly extended ; so a compilation can be used as a new boot.img/boot.dict if required. At a later date the dictionary could be imported into the image itself to give a fully or partly interactive system.