**MZ Programming Language / System**

Every word executes immediately. The responsibility for code generation belongs to the word itself. All words have an associated value with is a 24 bit page/address value and a “type” which tells them what to do.

**Types of words**

Code words (Type 0)

Type 0 code words are similar to standard FORTH calls. A call to the address is generated when the word is called ; so ‘emit’ may generate “call $80F2” for example.

Constant words (Type 1)

Type 1 words are a bit like FORTH arrays and variables. The value in the address (the lower 16 bits) is pushed on the stack as if it were a constant.

Macro words (Type 2)

Type 3 words are like FORTH immediate words. The code routine itself is called, which may generate code. This is for words like *if* and also standard macros

It is not mandatory, but advisable for some words to be inaccessible via execution ; structure words, words manipulating the stack and so on.

**Executing vs Compilation**

When executing a word (e.g. it is in Yellow) it is done in a “wrapper” which loads the registers, does the execution which generates the code, and saves the registers afterwards. This will involve some hacking of the pointer used in byte and word compiles ; all this code is generated and then called so it works with Code, Address and Immediate words interchangeably.

**Language Functionality**

The basic model is two 16 bit registers known as A and B. The language is broadly similar to the original versions of ‘M’.

Everything is a word, with the single exception of integer constants, which are values in base 10 mod 65536. No hexadecimal constants are used. These are treated as a ‘special case’ viz. they generate the constant code, same as address words. Words are assumed to be “Code Words” and their execution/data address is the free address pointer ; these can be overwritten of course.

Constants operate as in ‘M’ in that a constant is prefixed by copying A to B. So 12 22 + will

1. Put the current value of A in B, and 12 in A
2. Put 12 in B, and 22 in A
3. Put 34 in A, leaving B unchanged.

The variable mechanism has changed. There now exists two macro words !! and @@ these examine the previous code to check it is a constant of some sort (if it isn’t , that’s an error) e.g. it compiled ex de,hl ; ld hl,$nnnn

The macro then modifies that code to be either *nop ; ld ($nnnn),hl*  (for !!) or *ex de,hl; ld hl,($nnnn)* (for @@)