**MZ Language**

MZ is a Z80 based FORTH derivative. The main problem with FORTH on a Z80 is the twin stacks ; a Z80 only has one stack, the return stack, so much of the work has to be done by one of the other registers which move data in 8 bits.

MZ solves this problem by replacing a stack with three 16 bit registers, known as A, B and C. These registers directly map (currently) to HL, DE and BC.

The MZ language consists of words like FORTH. Unlike FORTH, every word is treated identically, though they do not of course behave identically.

MZ code is simply a sequence of MZ words, or decimal constants

The decimal constants are put on the ‘stack’ using the sequence ex de,hl ; ld hl,<constant>.

**Words**

Words are straight Z80 code, ending in a RET (or sometimes a JP rather than a CALL/RET etc.).

All words are compiled when treated as source code. There are no exceptions to this – all words are effectively *immediate.*

The purpose of almost all words is to compile their functionality.

**Standard Words**

Most common words therefore start with the same piece of code. This is a call to a routine that compiles their functionality.

This routine simply pops the return address, which is the first byte of the code that makes up the routine, and compiles a CALL to that.

So, a routine which prints a star might look like this :

call <compilecalltofollowingcode>

ld a,’\*’

call printcharacter

ret

**Macro words**

Macro words, like standard words, compile their own functionality. However, they do it slightly differently.

For some actions, compiling CALL xxxxx is either too slow, wasteful of space, or sometimes both.

For example, the ‘+’ word is simply the Z80 opcode add hl,de which is a one byte single instruction. It is a bit wasteful to wrap this with a Call and a Ret.

So Macro words work like simple Assembler Macros. They are a block of code which is copied out.

The same idea is used ; the first instruction is a call to a function which does the compiling work, and it pops the return address to get the information. In this case, the first instruction is a byte nn where nn is the number of bytes to copy (in the lower 4 bits), the upper 4 should be masked off. The macro code itself follows.

e.g. for + we have

call <compilecopyfollowing code>

db 1

add hl,de

The add hl,de is the only thing that actually belongs to the macro.

**Immediate Words**

Immediate words just behave exactly as normal Z80 code, without the call prefix. They are used for things like if..then and other code generating words.

To differentiate their creation, there are two “colon definitions” : (which creates a standard word) and :: (which creates an immediate word). The difference is that : puts in the initial call to the compile-call-to-yourself routine.

**Variables**

Variables are defined using variable <x> and they have a slightly different prefix again. They generate the following

call <compilemyreturnaddress+1>

dw 0

The variable on its own pushes its own address onto A->B as for a constant ; the routine called does this.

**Constants**

Besides words, the language contains constants, which are decimal integers, possibly prefixed with a ‘-’. These produce code that load a constant.

When you load a constant into the register set it always goes in A, but before it does that, it ***transfers the contents of A into B***. So constant 3 actually generates this code

ex de,hl ; transfer A->B

ld hl,3 ; load 3 into A.

The reason for this oddity is that it allows the fluency of FORTH, MZ’s parent language. You can write , as you can in FORTH.

count @ 2 \* 3 + count !

and it will work just the same (though MZ can do it better). Where it falls apart is when you need to use three values ; because then you have to use C or the return stack as a working space. (words like a>b b>c r>c exist for this purpose).

When I first conceived of this I thought it likely it would lead to an explosion of variables, but actually it doesn’t, though there are certainly more variables.

**Language Environment**

None of this actually allows you to do anything interactively. This is achieved by when you wish to execute interactively, using the word to generate code in a separate designated buffer ; a <ret> is added to this, creating a callable routine (which is wrapped in code to load and save the A registers).

This means there are certain words that will crash the system. To help with this, certain words are deemed “protected”. These are macros that generate such code, things that pop on/off return stacks, and so on. These set the 7th bit of the number in the ld a,x (which is masked off). This can be used to generate an error if this code is generated to be directly executed.

**Private variables/words**

Any word which begins with an underscore is deemed private and can be removed from the dictionary when it is crunched.

Note that while routines are callable cross page, variables are not accessible cross page (unless they are in $8000-$BFFF). It should be the norm to declare variables private and provide accessor and mutator functions in modular code.

**Dictionary**

|  |  |
| --- | --- |
| Offset | Contents |
| +0 | Offset to next entry, normally word length + 5, 0 at end |
| +1 | Page number of word |
| +2,+3 | Address of word |
| +4 | Length of name |
| +5 | First character of word name (ASCII) |
| +n | Last character of word name (ASCII) |

**Core Words**

|  |  |  |  |
| --- | --- | --- | --- |
| Word | Com? | Pro? | Description |
| - |  |  | B – A → A |
| ; |  | Yes | Return from subroutine |
| : <name>  :: <name> | Yes |  | Define a word. |
| ! |  |  | 16 bit write B to memory locations A,A+1 |
| !! @@ && | Yes |  | Variable operators |
| .hex |  |  | Print word on console |
| [-]if .. then | Yes |  | Conditional execution |
| @ |  |  | 16 bit read of A → A |
| \* |  |  | A \* B → A |
| / |  |  | B / A → A |
| + |  |  | A + B → A |
| +! |  |  | Add B into memory at A,A+1 |
| < = > <> >= <= |  |  | Comparison B vs A → A = -1 if true, - if false |
| 0- |  |  | 2’s complement A |
| 0< |  |  | A is -1 if A <0, 0 otherwise |
| 0= |  |  | A is -1 if A == 0, 0 otherwise |
| 1+ 1- 2+ 2- |  |  | Increment/Decrement A |
| 2\* 4\* 8\* 16\* 256\* |  |  | Shift A left |
| 2/ 4/ 16/ 256/ |  |  | Arithmetic Shift A right |
| a>b b>a b>c etc |  |  | Register→Register transfer |
| a>r abc> b>r c>r |  | Yes | Push one or all registers on stack |
| abs |  |  | |A| → A |
| and |  |  | A & B → A |
| begin .. [-]until | Yes |  | Conditional loop |
| begin .. again | Yes |  | Unconditional loop |
| break |  | Yes | Compile a CSpect break |
| bswap |  |  | Swap upper/lower bytes of A |
| c! |  |  | Save B at memory location A (8 bits) |
| c@ |  |  | Load A from memory location A (8 bits) |
| copy |  |  | Move C bytes from B to A, copes with overlap. |
| debug |  |  | Display ABC on console |
| fill |  |  | Fill C bytes of memory with B starting at A |
| for .. next | Yes |  | Loop repeated a constant number of times. Note, 5 for next counts 4,3,2,1,0 and index in A at the start of every loop. |
| halt |  | Yes | Stop Z80 working |
| i | Yes |  | Get loop index into A, A → B |
| inkey |  |  | Read current pressed key or 0 if none |
| gfx.write.char |  |  | Print character E in colour D (Spectrum colours) at position HL |
| gfx.initialise |  |  | Initialise display mode, clear screen and border, home .hex cursor. Different kernels have different graphics support. |
| max |  |  | Larger of A,B → A |
| min |  |  | Smaller of A,B → A |
| mod |  |  | Remainder of B / A → A |
| module  endmodule | Yes |  | Brackets module code. All it does is throw an error if the module starts and ends in a different page of memory. |
| not |  |  | One’s complement A |
| or |  |  | A | B → A |
| p! |  |  | Write B to port A (16 bit address) |
| p@ |  |  | Read A from port A (16 bit address) |
| r>a r>abc r>b r>c |  | Yes | Pop one or all registers off the stack. |
| swap |  |  | Swap A and B |
| sys.info |  |  | System Information address → A, A → B |
| variable <name> | Yes |  | Make last definition a variable accessed via && !! and @@ |
| xor |  |  | A ^ B → A |