**About Z80 Color Forth**

This document describes the implementation of Color FORTH, as close as reasonable, on the Z80.

The basic problem is the Z80 itself. It has one workable 16 bit stack. FORTH, realistically, needs two or it throws away a lot of its advantages.

There are various ways round this. This works round it by avoiding recursion (except possibly to self) and using the stack primarily as a data stack. Despite this, the code is subroutine threaded.

For example, a standard call and return is coded like this ; it’s very like subroutines used in computers like the PDP-8 and CDC160 which don’t have stacks.

|  |  |  |  |
| --- | --- | --- | --- |
| call routine | >>> | pop hl | *get return address* |
|  |  | ld (exit+1),hl | *overwrite the empty jump at the exit* |
|  |  |  |  |
|  |  | (code) | *word code … sp is the same as at call* |
|  |  |  |  |
| continues | <<< | jp $0000 | *jump to the return.* |

This manages a reasonable combination of limited overhead and minimal code bloat.

In words that don’t call other words the process can be simplified to one of these three.

|  |  |  |
| --- | --- | --- |
| **HL not used** | **IX not used** | **Stack doesn’t change** |
| pop hl | pop ix |  |
| (code) | (code) | (code) |
| jp (hl) | jp (ix) | ret |

The last can be used for unary or words that take no parameters at all, so that the stack does not move during the routine.

For the exit , we use the ;s word, because ; on Color Forth closes If..Then so will be delayed until if…then is written. Multiple ; are allowed, the first one creates the closing jump, and its address is available to any subsequent jumps.

In our system the head of the data stack is cached in DE. This allows code generation to be simpler.

|  |  |
| --- | --- |
| **Cached increment** | **Uncached increment** |
| Inc de | Pop de |
|  | Inc de |
|  | Push de |

Besides this obvious improvement, the routine does not modify the stack, so if you wanted it as a word, then it just needs a ‘ret’ rather than one of the other options.

**Generated Code**

Code is generated for the core vocabulary from files in the ‘vocabulary’ subdirectory. These are marked with @macro,@forth or @both, followed by the name. This indicated where the code goes.

The actual name is encrypted using its ASCII values with either .f or .m appended. It is the prefixed with ‘word\_’

@macro

Puts the code in, but precedes it with a routine to automatically copy it ; so you might define ‘@’ as a macro, which would generate ex de,hl ; ld e,(hl) ; inc hl ; ld d,(hl). Macros go in the FORTH dictionary as they are executed at compile time (usually).

@both

This is like @macro, except a second word is generated which wraps the code in the pop hl ; ld (xxxx),hl ….. jp 0000 wrapper

@word

This is a standard word, which can be wrapped however you want ; HL, IX or the JP method.

Words that are created this way include (at the time of writing) :

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ! | \* | + | +! | - | , | ;s | / | /mod | @ | 1, | 2\* |
| 2/ | abs | and | b! | b@ | bswap | debug | digit | drop | dup | fill | forth |
| h | here | less | macro | max | min | mark | empty | max | min | mod | move |
| negate | nip | or | or! | p@ | p! | pop | push | top | u+ |  |  |

**Compilation etc.**

The word compiler, which takes a tagged word (begins with $81-$87, ends with anything with bit 7 set) has to be re-entrant. This causes problem with the stack, because the stack is used during the compilation.

So a 2nd simpler stack is used based around IX. This allocates 64 bytes for each depth of re-entrancy (allow for 3 ?) and works like a mini stack.

On entry to the compilation routine, the stack is decached (e.g. DE is pushed on it) , and it is stored at (ix+0,1) and the stack pointer is set to IX+62.

On exit the stack is reloaded from (ix+0,1) and recached (e.g. DE is pulled off it).

Note that when words are executed, IX may changed, so reload it afterwards. IX can also be reset by the outer command line.

**Colours and their function**

Red Words ($82)

Red words create a new dictionary in either FORTH or MACRO and a new calling entrance code (pop hl ; ld ($0000),hl) having saved the start of the code. If the previous code wasn’t closed – this saved value is non-zero, an error occurs. It also zeros the close-code address

;s reverses this ; it checks for the start value being non-zero ; if it is, it zeroes it and patches the ld ($0000),hl and compiles the jp (hl) if the close-code address is zero ; if it’s non zero it simply jumps to it, allowing multiple exit points.

Magenta Words ($83)

Magenta words when executed (they are always MACRO) push their own address on the stack. This puts a routine which compiles the address of the word immediately following the magenta word e.g.

call CompileCodeThatPushesAddress

dw $0000

The address pusher compiles push de ; ld de,xxxx which pushes the address as a constant. This constant is also saved so this can be overwritten by !! and @@

Green Words ($84)

Green words first check MACRO, if the word is present it is executed. They then check FORTH, if the word is present a call is compiled to it. If that fails it looks for a constant (negative postfix type)

Cyan Words ($85)

Cyan words look in Macro, if the word is present a compile is called to it.

Yellow Words ($86)

Yellow words look in FORTH, if the word is present it is executed. If not, it is checked to see if it is a constant, if so it is pushed on the stack.

White Words ($87)

White words are comments. They are not in compiled buffers, but can be in normal ones.