Z80 Color Forth

This document describes a Z80 based Color Forth designed to run on the Spectrum Next, or indeed other Spectrum variants.

As with other Color Forths, the core is written in local assembler. The bulk of Color Forth is ‘created’ via the SNA format. Words are added by being translated from a text file into 2+6 bit format code (this is where the upper 2 bits define the colour, Red/White/Green/Yellow and the lower 6 bits are the ASCII value). Comments are removed, and words deemed private are replaced with \_\_x where x is a-z (they are not kept in the dictionary)

This is then stored in memory from $4000-$59FF – the Spectrum screen area. When the system boots, it runs by compiling code in this space ; the resulting file is the manually saved as a .SNA file.

This can be repeated if necessary ; it may well not be as this is about 7k of code with no comments and often shrunk dictionary entries.

At higher levels (e.g. Spectrum Next) code can be loaded and compiled using the Next OS into paged memory.

Dictionaries

There are two dictionaries ; the standard dictionary and the macro dictionary.

Words in yellow (immediate) can only run from the standard dictionary ; Words in green (compiled) can be run from either (macro is checked first) ; the macro dictionary entries are executed, the standard dictionary compiled as calls. The dictionary grows downwards in contended memory.

Implementation Notes

This implementation uses the pop and save method to avoid the two stack problem. This means words are not re-entrant and not recursive except for self which is a synonym for call [self] ; ret and actually just does a jump to the start of the word.

The pop and save method involves words generated doing a pop hl ; ld (exitjump+1),hl at the start and jp $0000 at the end, the address being patched by the ld (),hl instruction. Words are thus called as normal, but are only briefly on the return stack which is used as a data stack.

This also means words cannot fall through as is common in ColorForth with its separate dictionary.

Register DE is used as the top of stack value, and the return stack is the rest of the stack.

Assembled words

Assembled words can use the same pop&save mechanic or simpler ones pop hl …. Jp (hl), or if the stack depth does not change, nothing at all.

Assembled words are written in Z80 assembler, and are kept in a subdirectories. Anything in a subdirectory which contains a ‘fast’ directory is deemed to be placed in memory from $8000, anything else is placed before $8000 ; thus some code can be put in quicker memory if needed.

Source files can be named anything, but should have a comment ;; @ <name> which indicates that the following is code for <name>. There can be multiple words in any single file. The comment is replace by a star barred comment (for source/listing readability) and the label definition\_ followed by a sequence of lower case ASCII codes encoded in 2 digit hexadecimal form separated by \_

So “ace” would be encoded as label definition\_61\_63\_65.

All these definitions would go in the standard dictionary

Source code

Source code uses a more standard FORTH format, it is black and white, uses : x for colon definitions ; words are in Green normally unless surrounded by curly brackets. Comments are done using // and are translated out.

Memory map

|  |  |  |
| --- | --- | --- |
| **Address range** | **Contents** | **Usage** |
| 0000-3FFF | Spectrum ROM | Sinclair ROM Image, only used for Font. |
| 4000-57FF | Display RAM | Buffer for compiled code |
| 5800-59FF | Attribute RAM | Buffer for compiled code |
| 5A00-5AFF | Attribute RAM | Stores temporary definition addresses \_\_a a $5A00, \_\_b at $5A02 etc. |
| 5B00- | Contended RAM | Boot code |
|  | Contended RAM | Slow words that can run in contended memory |
|  | Contended RAM | Data areas |
|  | Contended RAM | Dictionary (grows down) |
| 8000- | Fast RAM | Fast words that benefit from uncontended memory |
|  | Fast RAM | New compiled code routines |
| C000- | Paged RAM | Compiled code that may be paged on Next/128 versions |

Compiler code

The compiler looks at each word in turn

* White words are ignored
* Red words create a dictionary entry and the prefix ; the prefix can be unwound for things like *variable* and replaced with appropriate code.
* Green words are compiled using the macro dictionary (word is executed) or standard dictionary (word is called)
* Yellow words are always executed but must be in the standard dictionary.

The compiler code has its own local stack. The reason for this is so it can have multiple levels of calling etc ; when it wants to execute a word in some way it can reload the stack and DE, execute the word and the save the stack and DE. The real stack/DE is saved on entry and restored on exit.