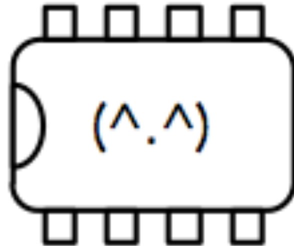
4th Saga ch2
s4th Origin Story



www.ZENMCU.com

Draft 1 2023-02-07

Copyright © 2023 by Trenton Henry, All rights reserved

s4th

This is the story of the origin of s4th. I am only writing this down for myself, so that maybe I can learn something about my design process. I do tend to think things through before I start coding, but s4th involved far more thinking through-ing than normal, and I have been curious about why that was the case. I wasn't struggling with technical issues, I was struggling with product design issues.

It's no secret that while I am making toys to build my own projects I do have a hope of someday selling some to offset the costs of building them. The ZENMCU boards are a case in point. What is the best PCB footprint? Will these be mounted on breadboards or soldered into projects? Should they lie flat, or stand on their edge? And all manner of other, generally pointless at this stage, concerns.

So the rest of this article is a reworked resurrection of my notes and thoughts on what s4th needs to be and why, generously edited to make me look less like an idiot. And, by the way, s4th is "spanda 4th" because ... its for my i/o expansion cards..

The purpose of x4th is to enable me to develop code for my gadgets without needing to use debugger pods. Plug it in, load up x4th, write code, download, unplug and go. However, that does require running an application on the host, namely x4th (whether you use the GUI or not). And that's fine. I designed it for how I imagine I want to sue the ZENMCU boards.

But Spanda cards (the add-on modules that attach to ZENMCU boards to provide useful i/o) are intended to be "fixed function" devices. Maybe think of them as I2C devices, but with some smarts and a UART interface so you don't have to write a driver for them. In a sense they are very vaguely like the Apple II add on cards which carried their own ROM to be executed by the main CPU. Except, Spanda cards have their own CPUs, so the base board only needs to send simple commands over the UART. Maybe its reminiscent of vintage communication devices with an AT command set.

So how should the "host interface" of Spanda card actually work?

I2C has a lot to offer, but there are limitations on addressing multiple alike devices, complete inconsistency on how to read/write registers in such a way as to make them do the appropriate thing, etc. Each one needs a new custom blob of code to access it.

Other examples include Txtzyme (link), Simpl (link), Stable (link), USB BitWhacker (link), and probably others. Each of these provides an arguably convenient host interface with enough functionality to allow delays and loops and in some cases very simple functions. And they are pretty neat, really. You can do interesting things with them, and they seem simple to implement. I looked to them for ideas and inspiration.

A wide variety of approaches to system decomposition and implementation strategy has been applied to Forth. These all depend on the requirements of the system as a whole. (See com.zenmcu.0041-0_(a-taxonomy-of-forths-1).pdf for additional discussion.)

The purpose of s4th is to provide a mechanism for a 'host' system to issue commands to, and receive responses from, Spanda cards. Conceptually it is similar to I2C, but it is not primarily based on reading and writing registers. Spanda is an abstraction between hardware specific interfaces (I2C, SPI, GPIO, ...) and the 'host', where generally the host is some microcontroller such as a ZENMCU board, or Arduino. A host could also be an Rpi, or PC, but those are not resource constrained and so can afford to implement multiple different drivers for every attached i/o device.

At the bare minimum a Spanda card must be able identify its product id and version, and provide a simple way to access the functionality of the card. For example, an RGB LED Spanda card should at least allow setting the R, G, and B values, as well as reading back the current values.

The most obvious way to do this is with the setter words R!, G!, and B!, and the getter words R@, G@, B@.

Another option is to tread R, G, and B as variables, with a word like UPDATE to cause the current values to take effect.

Yet another option is to tread R, G, and B as monitored variables. I.e, the system polls to see when they have been updated and acts accordingly.

It might also be nice to be able to have the LEDS blink or fade according to some pattern. Maybe fade from this RGB to that RGB in this many seconds. Or periodically blink with attack, sustain, and delay times. For that type of activity it is necessary to provide the ability to loop and delay, possibly variables for counters, and maybe even subroutines.functions.

Therefore s4th takes the approach of providing a variety of builtin functions, and a RAM space for user programs. (This is very similar in concept to the Basic Stamp, except its Forth-like, rather than Basic-like). And there is the ability to save the contents of the RAM into flash, and restore it later so that user programs persist across resets and power cycles.

Ok, but why is s4th talking to the host in text? I.e., why does it expect you to type 255 as 3 separate characters 2, 5, 5, as text instead if in binary? Why does it respond with test instead of binary? Isn't that a less efficient use of the bandwidth of the host interface?

Well, yes, it is. And I may add a binary mode if it becomes an issue. But for now text is easy to see on the wire, interpret, and debug by humans. I.e, by me.

In any case, the memories of the ZENMCU microcontrollers are very small, and there is a limit to what can be included as builtin primitives. Loops, timers, and user functions seem like the minimum needed. Anything beyond that, like exceptions, aborts, <builds does>, and even possibly decompiling, are luxuries for Spanda cards. (I admit that decompiling is extremely useful when I am debugging this.)

And I am, as of 02/12/23, at a crossroads. I have implemented ~90-ish% of the x4th code library as s4th primitives, and have run out of code space. Without USB. With optimizations. So I am at the point where I must give each word, or set of related words, careful consideration. Only the most necessary words can be included.

These are, obviously, the product specific words (like R, G, B etc), the ability to define new words, conditionals, iteration, variables, constants, and timing. I want it to be at least as capable as Txtzyme et al that I mentioned previously. It doesn't need to be the ultimate Forth system.

I have convenience, compiling, debugging, exception handling, etc primitives which aren't necessary for Spanda cards. They are nice to have for a more capable processor such as the SAMD21. So it may be time to conditionally compile, or decompose, s4th along product lines.

And now it is 02/14/23. I have continued to putter about with s4th, with reasonable success. But I am up against the hard limit of 12KB of code space available for the VM and

core library of builtins, which is both good and bad. It is good in that it forces me to examine the code in detail in an effort to optimize at least the low hanging fruit. But it is bad in that it distracts me from the primary mission of building ZENMCUs, Spanda cards, and my robot society by sending me down the optimization rabbit hole.

But, like x4th before it, s4th fits a niche. It is serviceable, though of course needs polish, and is quite cramped, on the SAMD10/11C parts with 16KB of flash and 4KB of RAM. But it likely would be better suited to the SAMD21E with 256KB of flash and 32KB or RAM. So I will keep it on hand and continue to develop it. And I will back-port some of its improvements to x4th. But I will continue to think about some new Forth-like which is a better fit for the SAMD10/11C.

And ... one Idea I've been mulling is a sort of a hybrid between x4th and s4th. Because x4th is cross compiled and has no local dictionary it occupies far less of both types of memory on the target. The x4th interpreter is just 'read a number', 'read a command', and do it, where commands are # to push it, or x to execute the number on the top of the stack.

But x4th, at least on the host, is complicated in that it allows compilation into host memory (at uphere) and into target memory (at dnhere), and downloads/uploads to/from the target. It models the memories of both sides of the USB and manages to synchronize them. Granted most of the complexity is on the host, but it is also on the developer (aka myself) as one must keep track of what one is doing in order to avoid compiling to the wrong memory.

So I am imagining a sort of hybrid with the dictionary in host RAM containing the primitives and the user defined words. Like x4th, and unlike s4th, the dictionary and the code memory are separate. Like s4th, and unlike x4th, the target code is compiled into RAM, not NVM. This provides 12KB of NVM for the VM and core, and 4KB of user code. The remaining 4KB of NVM is for saving/restoring the RAM.

This has x4th's advantage needing less C code on the target, and s4th's advantage of avoiding constantly erasing/writing flash. It may be a reasonable blend of the benefits of both styles. But it does have the disadvantage of requiring a special purpose app on the host (the whatever4th front end). I.e., you can't just plug it into USB and open picocom on a serial port. Instead you plug it into USB and open whatever4th.

Such a hybrid seems like it would be a good mixture of x4th and s4th. The flash is all for primitives, except for the save block, and the RAM is all for your application. The interpreter is simple and lightweight. If I implement it with some thought it might be possible to set it up such that all of the compilation time activities happen on the host and all of the execution time activities happen on the target. So you have to have the target connected to do anything. Maybe that is unwise. Maybe it is better to mirror all RAM writes to the target if it is connected, but allow simulation in its absence.

I like x4th's file arrangement so I may reuse that. Or I may try do different this time. Combining everything into one file does simplify quite a bit, but it can get difficult to find things, and difficult to track change meaningfully in version control.

I like s4th's generated dictionary etc, so reuse that. But don't compile user words into a separate ramdict. And only compile the TOK/CFA in to the dict, not the code, x4th style.

This gives me one compact dictionary in host RAM.

I also like s4th's simpler view of dm and dc, wherein things aren't all columnar like in x4th. It seems more Forth-esque.

I also like s4ths simpler execution model which only needs ip (not wp) and does not require storing the tokens of the builtins in the code space (which wastes RAM).

This model could be made into a 32 bit machine by storing C function addresses is XTs and RAM addresses as CFAs and checking the range to know which one. Doing that would allow addressing physical memory and registers, as well as eliminate the table of builtins[].

Akshewally ... I might be able to eliminate that from s4th already...? TOKs are 16 bits. Negative TOKs (hi bit set) are XTs (indices into builtins[]) and positive TOKs (hi bit clear) are CFA's (indices into vm.ram[]).

The highest flash byte address on the SAMD1xC14 is 0x3FFF. With the hi bit set that's 0xBFFF. So I can actually store a 'pointer' to flash as an XT instead of an index if I clear the hi bit and load it into a U32 / pointer. That would eliminate 608 bytes (5%) of the s4th flash.

Of course, there is no way to generate a dictionary with those XTs baked in since they aren't known until link time. And that won't work on the host side for a simulator since its function pointers are 64 bits. Regardless, that means that the dictionary has to be resolved at compile time, which means baking in only portions of the function pointers with bits twiddled, and I am not convinced that can be done at compile time as it makes the initializer a non-constant. So it has to be done at runtime. And this is almost certainly more code wasted doing that than is wasted by the table of builtins[]. I didn't measure it, but it does seem obvious.

So the next possibly clever thing is to shrink the size of builtins by half by putting the munged pointers into it instead of full pointers. But that has the exact same initialization issue as the half pointers in the dictionary.

And a switch/case is basically the same as my builtins[] table, if the compiler is competent, so that can't possibly be a win. So the compressed pointers idea for s4th seems like it's not doable at compile time.

It is possible, though I doubt it's worth doing, to take the compiled binary which has FFFF place holders in the dictionary, and patch in the munged pointers as a post process if I harvest them from the map file, or using binutils. But I mean ... that is getting a bit desperate.

It would indeed save 608 bytes of flash doing that (with the current number of builtins) but what is the next optimization when that fills up? Probably that's not worth doing, honestly, except possibly as a last ditch measure. Because the next step is to remove the dictionary all together which becomes the hybrid x4th/s4th offspring I mentioned previously.

So ... maybe the hybrid is the next thing to build. It is still a distraction from the primary mission, but somehow I feel that it needs doing. This is why I never actually get to the point of having a product...

End