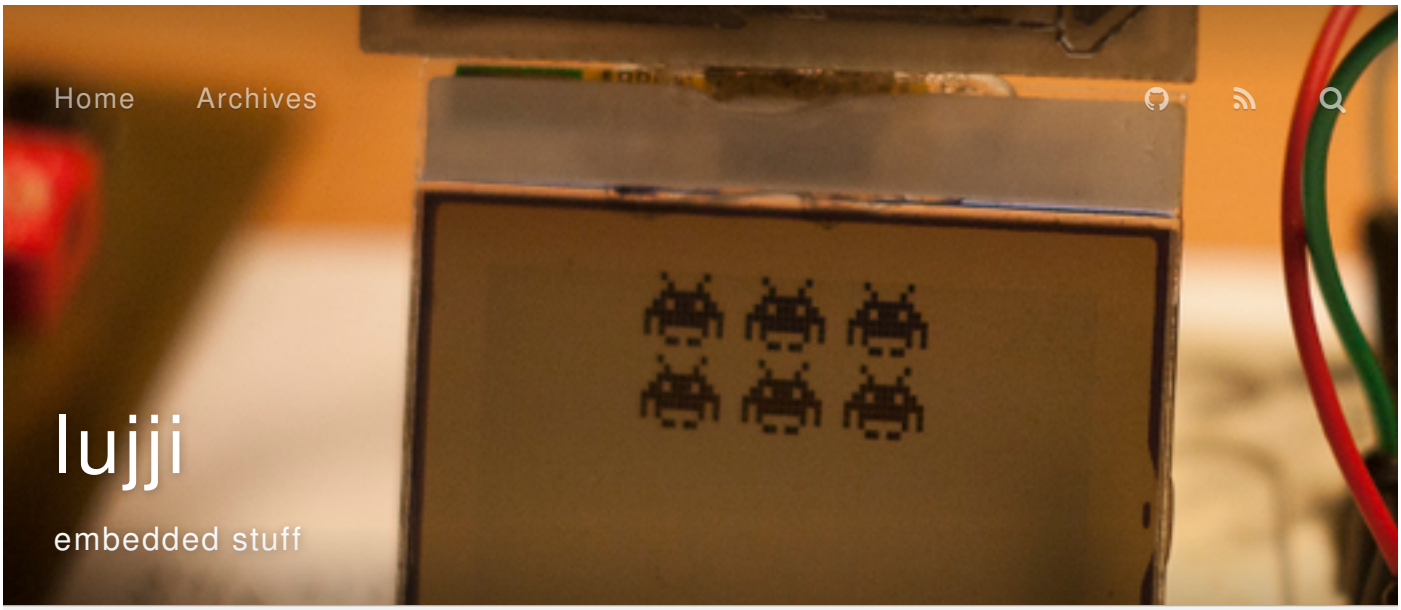

2017-07-26

Executing code from RAM on STM8

A short article where we investigate how executing code from RAM can be achieved on STM8 with SDCC toolchain.

All right, I've been avoiding this topic for quite a while, so I wanted to deal with it first before finishing other articles. The reason for me to avoid this topic was mostly because I needed to come up with a relatively clean solution that would be worth writing about. I had an assumption that SDCC was not the right tool for the job, and some of the hacks that I came across while researching this topic only made this assumption stronger. But I'm more than glad to say that I was wrong.

Overall the mechanism for copying functions into RAM is not complicated: you place your function in a separate code section, reserve some memory for this function and finally, copy the contents of this section into RAM. The hardest part is to figure out how to accomplish all that with SDCC toolchain. Let's find out.

First of all, SDCC port for STM8 supports `--codeseg` option, which can be also invoked via a pragma. In order to place a function into a specific code section we have to implement this function in a separate .c file, compile it and link with our application. For this example we'll take a function that sends a null-terminated string over UART:

```
1  #pragma codeseg RAM_SEG
2  void ram_uart_puts(const char *str) {
3      while (*str) {
4          UART1_DR = *str++;
5          while (!(UART1_SR & (1 << UART1_SR_TC)));
```

```
    }
}
```

After compiling the source we should be able to see `.area RAM_SEG` above `_ram_uart_puts` symbol in the output listing.

Now that we have a separate section containing a single function, we need to find some way of getting the section length in order to know how many bytes to copy. For that we'll resort to SDCC ASxxxx Assemblers documentation, which is an impressively large document, but don't worry - we'll only need small portions of it.

'General assembler directives' section tells us that assembler generates two symbols for each program area (code section): `s_<area>`, which is the starting address of the program area and `l_<area>` - length of that program area. Unfortunately, you can't access these variables directly from C. But you can access C variables from assembly, which means that retrieving code section length can be achieved with just a single line of assembly code:

```
volatile uint8_t RAM_SEG_LEN;

inline void get_ram_section_length() {
    __asm__("mov _RAM_SEG_LEN, #l_RAM_SEG");
}
```

Here I'm assuming that the function is small enough to fit into 255 bytes. If that's not the case, things become a bit more complicated:

```
volatile uint16_t RAM_SEG_LEN;

inline void get_ram_section_length() {
    __asm
        pushw x
        ldw x, #l_RAM_SEG
        ldw _RAM_SEG_LEN, x
        popw x
    __endasm;
}
```

We're using `ldw` instruction to load the section length into a 16-bit index register X, which is then copied into uint16_t variable RAM_SEG_LEN. Note that symbol names of C variables are generated with a leading underscore. Also note that the code snippet is surrounded with pushw/popw instructions - this is done to preserve the contents of register X since we don't want

our inline function to break any other code that might be using this register.

Now the last remaining thing is to copy the subroutine into RAM:

```
uint8_t f_ram[128];
void (*uart_puts)(const char *str);

void ram_cpy() {
    get_ram_section_length();
    for (uint8_t i = 0; i < RAM_SEG_LEN; i++)
        f_ram[i] = ((uint8_t *) ram_uart_puts)[i];
    uart_puts = (void (*)(const char *)) &f_ram;
}
```

Since there is no elegant way of getting code section length at compile-time, we simply declare an array of fixed size and make sure that it's large enough to store our RAM functions. SDCC does not support variable-length arrays, so we can't allocate this memory on the stack either. A nicer workaround would be to use `malloc()`, but it just feels wrong. We *could* of course reserve the exact amount of bytes in the .data section in assembly and declare f_ram as extern. But here's a thing about assembly: once you start optimizing things, it's really hard to stop. Quite often I come across some code which contains so much inline assembly that makes me wonder why the author bothered with C in the first place.

Keep in mind that some processor instructions can use both absolute and relative addressing, which might ruin your day when relocating functions with external dependencies, so make sure that you always check the listing. The general rule is: addressing within the function itself must be relative and accessing external symbols must be done via an absolute address. Minimizing external dependencies and keeping RAM functions compact and self-contained will definitely help preserving your sanity.

That's it for now. As always, code is on github.

#assembly #sdcc #stm8

Comments Share

NEWER POSTS

Mixing C and assembly on STM8

OLDER POSTS

Bare metal programming: STM8 (Part 2)

12 Comments lujji's blog Disqus' Privacy Policy 1 Login

Recommend 2 Tweet Share Sort by Best

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

مصطفى • 2 years ago

Thanks lujji, great work!
But I cant understand the reason why we need to put a function in RAM and then execute?

Reply • Share ›

Georg • 2 years ago • edited

hi lujji,

thanks a lot for your excellent tutorial! I really marvel at your minimalist approach :-) The RAM example worked out of the box and is easy to understand :-) However, I still have some questions / remarks:

1) do I understand correctly that this approach requires that the RAM routines don't use absolute jumps within RAM or calls between RAM routines...? Because the .map file shows e.g. RAM0 in flash at 0x81E6. If my understanding is correct, how do you ensure that absolute jumps/calls within RAM are not used by SDCC?

2) and if absolute jumps or calls within RAM are required for some reason, how about this (different) procedure:

a) link for RAM (`-Wl-bRAM0=0x00a0`) --> absolute addresses are ok, but in hexfile code is in RAM and is lost after reset (already verified)

b) in hexfile move code segment from RAM to flash address --> is not lost after reset, but located at wrong address. I plan to add this "move" feature to

see more

Reply • Share ›

Rowan Crowe • 3 years ago

About a year ago I bought a batch of 20 of those little STM8 boards, and apart from making one of them blink I haven't done anything further. I had some vague plans to wire them together in a network to experiment with artificial life, with each node self-modifying its code through mutation, neighbours peeking and poking each other's memory etc. Coming across your article has moved that along, because executing native code (from RAM) would be so much simpler than having to code

work! Thanks for this article, and your other STM8 tutorials.

Reply • Share ›

lujji Mod ➚ Rowan Crowe • 2 years ago

Sounds interesting, any progress on your project?

11 Reply • Share ›

Анна Макарова • 4 years ago

Hello, Lujji,

You are my hero as always, thanks for the post!
Now we can (boot)load more STM8 code from external sources live, store it packed or encrypted there, make self-modifying code, debug parts of the code changing it on-the-fly and trying differing things even without uC reset!
Your posts make bare-metal mure elegant and appealing, than huge, slow and buggy high-level competitors/

Keep your great work on, your projects bring much more value to the one cheapest uC avaliable!

Reply • Share ›

lujji Mod ➚ Анна Макарова • 4 years ago • edited

Hi, glad you're still around. I'm planning to cover the bootloader subject in a short time.
By the way, there is Forth port for STM8 that could probably do everything you mentioned above.

Reply • Share ›

Alan • 4 years ago

Hi Lujji,
Amazing post! Please create more posts like this!
BR,
Alan

Reply • Share ›

lujji Mod ➚ Alan • 4 years ago

Glad to hear that. I'll be doing a few similar posts shortly.
By the way, I've been looking into NuttX recently thanks to your tip. Surprisingly, there seems to be a port for a fairly recent esp32, but no esp8266 port for some reason.

Reply • Share ›

Alan ➚ lujji • 4 years ago

https://www.youtube.com/c/N... BR, Alan

Reply • Share ›

lujji Mod Alan • 4 years ago

Thanks, I'll definitely give it a try. It would be interesting to see how NXFFS compares to SPIFFS.

Reply • Share ›

Alan lujji • 4 years ago

Very nice! Also there is the SmartFS on NuttX. This file system is very robust, it was implemented by Ken Petit a NuttX developer. Samsung extended the SmartFS to support journaling in their TizenRT (a NuttX's fork), but then changed the license from BSD to Apache, then people are afraid to incorporate it back to NuttX. BR, Alan

Reply • Share ›

ARCHIVES

RECENT POSTS

Executing code from RAM on STM8
Bare metal programming: STM8 (Part 2)
Bare metal programming: STM8

lujji at protonmail com