

# lujji

2017-08-08

## Serial bootloader for STM8

This article will cover developing a serial bootloader for STM8S microcontrollers.

### Contents:

- [Entry condition](#)
- [Serial protocol](#)
- [Flash block programming](#)
- [Interrupt vector table relocation](#)
- [Squeezing the last bytes](#)
- [Benchmarking](#)

### Entry condition

Bootloader code gets executed first, so we need some mechanism to decide whether we want to update the firmware or execute main application. These are the most common approaches:

- **Configuration byte:** an indicator flag, which is written by the application when firmware update is requested and cleared by the bootloader once firmware update is performed
- **Timeout:** the bootloader waits for some external event during start-up. When specified timeout is reached, main application is executed
- **External jumper:** external switch or jumper which selects between bootloader and application mode

We'll go with the third option, since it's the easiest one to implement in my opinion. We also need to

define where the main application will reside. Let's be generous at first and dedicate 1k of flash to the bootloader, although we'll cut the size down eventually. Since flash memory is mapped to address 0x8000, the destination address for the application will be 0x8400.

```
1 void main() {
2     BOOT_PIN_CR1 = 1 << BOOT_PIN;
3     if (!(BOOT_PIN_IDR & (1 << BOOT_PIN))) {
4         /* execute bootloader */
5         // ...
6     } else {
7         /* jump to application */
8         BOOT_PIN_CR1 = 0x00;
9         __asm__("jp 0x8400");
10    }
11 }
```

Our main application has to be compiled with `--code-loc 0x8400` option, which instructs SDCC where code should be placed.

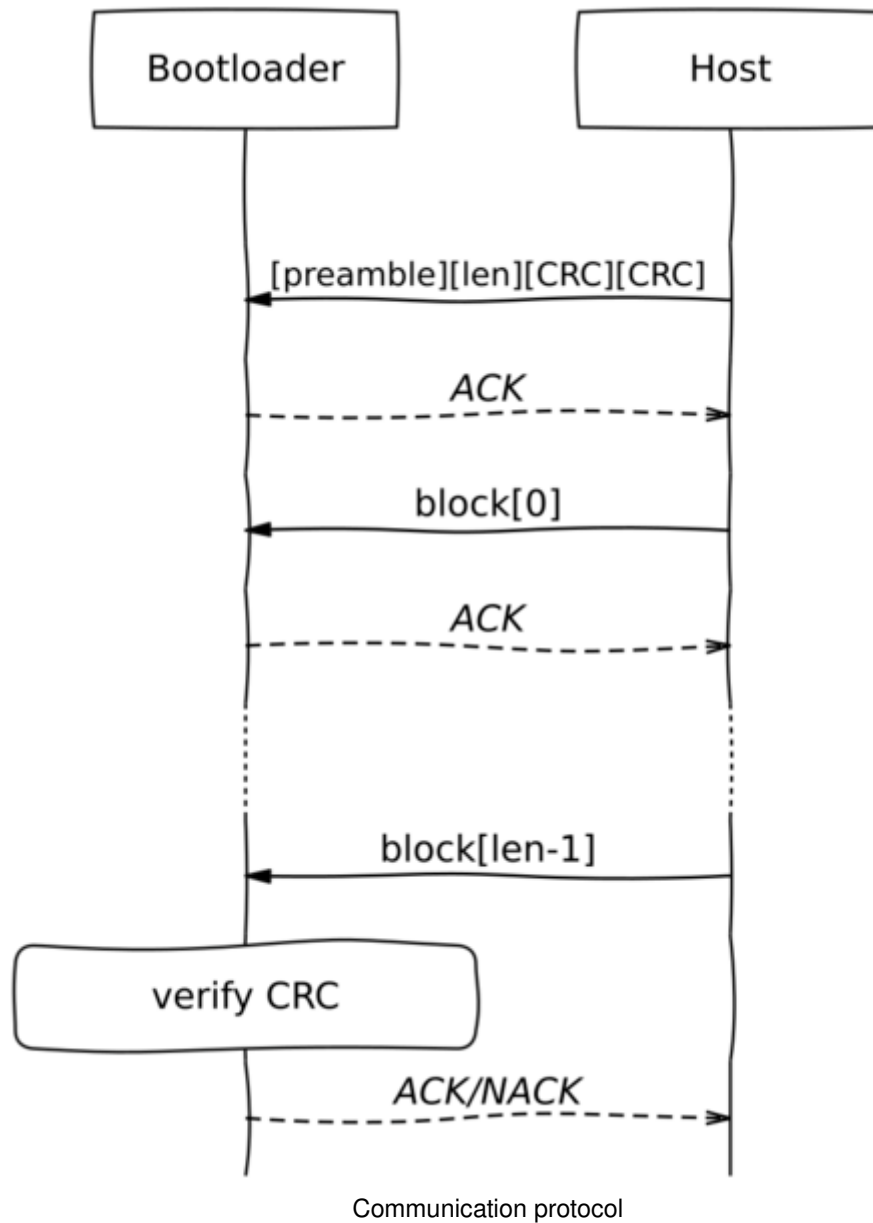
## Serial protocol

The firmware update will be initiated by sending the bootloader a preamble of 4 bytes. Preamble detection will look like this:

```
1 inline void bootloader_enter() {
2     uint8_t rx;
3     for (;;) {
4         rx = uart_read();
5         if (rx != 0xDE) continue;
6         rx = uart_read();
7         if (rx != 0xAD) continue;
8         rx = uart_read();
9         if (rx != 0xBE) continue;
10        rx = uart_read();
11        if (rx != 0xEF) continue;
12        return;
13    }
14 }
```

Once preamble is detected, the bootloader reads next 3 bytes: number of data blocks to be sent and two CRC-8 bytes which are duplicated to avoid transmission errors.

After that, we follow a simple request-response protocol: the host waits for an acknowledgment and then sends a fixed size chunk of data. Bootloader receives the chunk, writes flash memory block and sends ACK again to indicate that it's ready for another packet. When all data chunks have been sent, the bootloader verifies CRC and sends another ACK if CRC matches or NACK if it doesn't match. If the last chunk is smaller than defined block size, the remaining bytes are padded with 0xFF by the host.



Main bootloader code will look like this:

```
1  #define BLOCK_SIZE      64
2  #define BOOT_ADDR      0x8400
3
4  void bootloader_exec() {
5      uint16_t addr = BOOT_ADDR;
6      uint8_t chunks, crc_rx;
7
```

```
8     bootloader_enter();
9     chunks = uart_read();
10    crc_rx = uart_read();
11    if (crc_rx != uart_read())
12        return;
13
14    /* get main firmware */
15    for (uint8_t i = 0; i < chunks; i++) {
16        serial_read_block(rx_buffer);
17        flash_write_block(addr, rx_buffer);
18        addr += BLOCK_SIZE;
19    }
20
21    /* verify CRC */
22    if (CRC != crc_rx) {
23        serial_send_nack();
24        return;
25    }
26
27    serial_send_ack();
28 }
```

The microcontroller that I'm using is a low-density STM8S003F3 so the block size will be equal to 64 bytes. For medium and high density devices the block size is 128 bytes.

In one of the previous [articles](#) I mentioned that we can write flash one byte at a time, however on the hardware level one word (4 bytes) will be overwritten. Flash controller simplifies it for us by reading, modifying, erasing and writing a word each time a byte write is requested. We'll try to speed things up a little by writing 4 bytes at a time, which can be achieved by enabling `WPRG` bit in Flash Control Register 2. This bit is reset after programming is done so it has to be manually re-enabled before each write operation.

```
1 void flash_write_block(uint16_t addr, const uint8_t *buf) {
2     const uint8_t *end = buf + BLOCK_SIZE;
3     uint8_t *mem = (uint8_t *) addr;
4
5     /* unlock flash */
6     FLASH_PUKR = FLASH_PUKR_KEY1;
7     FLASH_PUKR = FLASH_PUKR_KEY2;
8     while (!(FLASH_IAPSR & (1 << FLASH_IAPSR_PUL)));
9
10    for (uint8_t i = 0; i < BLOCK_SIZE; i += 4) {
11        /* enable word programming */
12        FLASH_CR2 = 1 << FLASH_CR2_WPRG;
```

```
13         FLASH_NCR2 = ~(1 << FLASH_NCR2_NWPRG);
14         *mem++ = *buf++;
15         *mem++ = *buf++;
16         *mem++ = *buf++;
17         *mem++ = *buf++;
18         while (!(FLASH_IAPSR & (1 << FLASH_IAPSR_EOP)));
19     }
20
21     /* lock flash */
22     FLASH_IAPSR &= ~(1 << FLASH_IAPSR_PUL);
23 }
```

There's one problem with this code though - it's incredibly slow. I was hoping we could get away with word programming, but clearly this is not the case.

## Flash block programming

The most efficient way of programming flash is the block programming method. The only downside is that the processor will no longer be able to fetch instructions from flash during programming, so we'll have to execute our code from RAM. I've already covered [executing code from RAM](#) before, so I won't go into much detail here.

Let's re-implement our flash programming routine:

```
1  #pragma codeseg RAM_SEG
2  void ram_flash_write_block(uint16_t addr, const uint8_t *buf) {
3      const uint8_t *end = buf + BLOCK_SIZE;
4      uint8_t *mem = (uint8_t *) (addr);
5
6      /* unlock flash */
7      FLASH_PUKR = FLASH_PUKR_KEY1;
8      FLASH_PUKR = FLASH_PUKR_KEY2;
9      while (!(FLASH_IAPSR & (1 << FLASH_IAPSR_PUL)));
10
11     /* enable block programming */
12     FLASH_CR2 = 1 << FLASH_CR2_PRG;
13     FLASH_NCR2 = ~(1 << FLASH_NCR2_NPRG);
14
15     /* write data from buffer */
16     while (buf < end)
17         *mem++ = *buf++;
18
19     /* wait for operation to complete */
20     while (!(FLASH_IAPSR & (1 << FLASH_IAPSR_EOP)));
```

```

21
22     /* lock flash */
23     FLASH_IAPSR &= ~(1 << FLASH_IAPSR_PUL);
24 }

```

We use 'standard' mode which erases the block automatically. Some inline assembly is required to retrieve code section length:

```

1  volatile uint8_t RAM_SEG_LEN;
2
3  inline void get_ram_section_length() {
4      __asm__("mov _RAM_SEG_LEN, #1_RAM_SEG");
5  }

```

This trick works because RAM function is small enough to fit into 255 bytes - otherwise we'd use a slightly different mechanism. Finally, we copy the subroutine into RAM:

```

1  static uint8_t f_ram[128];
2  static void (*flash_write_block)(uint16_t addr, const uint8_t *buf);
3
4  inline void ram_cpy() {
5      uint8_t len = get_ram_section_length();
6      for (uint8_t i = 0; i < len; i++)
7          f_ram[i] = ((uint8_t *) ram_flash_write_block)[i];
8      flash_write_block = (void (*)(uint16_t, const uint8_t *)) &f_ram;
9  }

```

## Interrupt vector table relocation

An interrupt vector table (IVT) is a chunk of address space. Each entry in the interrupt table is called an 'interrupt vector', which points to the address of an interrupt service routine (ISR). When interrupt occurs, CPU registers are pushed on the stack, program counter gets set to the address of the corresponding interrupt vector and the first instruction at that address is fetched. There is a dedicated INT instruction which jumps to the interrupt service routine address. After the ISR finishes, IRET instruction must be executed in order to restore contents of the registers.

STM8 has 32 4-byte interrupt vectors starting at address 0x8000: RESET, TRAP, TLI and up to 29 user interrupts specific to each part. Immediately we start to see a problem: if the IVT is located at the beginning of the flash memory, which is where our bootloader resides, how is the main application going to handle interrupts? There are different ways to address this issue, but most of the time it boils down to something called 'IVT relocation'. Essentially we are going to have 2 separate vector tables for the bootloader and main application. The IVT inside the bootloader will

simply point to the corresponding vectors in the main application.

Let's illustrate all of the above on a random interrupt handler:

```
1 void tim4_isr() __interrupt(TIM4_ISR) __naked {  
2     __asm__("jp 0x8464");  
3 }
```

The interrupt handler is declared with `__naked` attribute, which instructs SDCC to omit `reti` instruction at the end of the handler, thus saving some program space. We can do so without any consequences, since we're jumping to another interrupt handler which will execute this instruction anyway.

I don't like this approach, and here's why. When interrupt occurs the CPU pushes registers on the stack, which takes 9 cycles. Then `int` instruction is executed (2 cycles) which jumps to our interrupt handler. Our interrupt handler performs a jump (2 cycles) to the application interrupt vector, which executes another `int` followed by the interrupt handler code (?? cycles) followed by `iret` (11 cycles). That's an overhead of 26 CPU cycle minimum, where 4 cycles were introduced by our interrupt handler. We also waste about 3 bytes of flash memory per handler. As a result, we end up with vector table and interrupt handlers that pretty much do nothing but consume space and processor cycles.

There is another approach: we can simply overwrite the first two blocks of memory with the application's IVT. If we do that, however, our main application will always be executed instead of the bootloader, since we've overwritten the reset interrupt. The solution is to skip the first two bytes, thus leaving the reset vector intact:

```
1 for (uint8_t i = 2; i < 2 * BLOCK_SIZE; i++) {  
2     *(uint8_t *) (0x8000 + i) = ivt[i];  
3     while (!(FLASH_IAPSR & (1 << FLASH_IAPSR_EOP)));  
4 }
```

The downside is that we can no longer use ST's User Boot Code (UBC) feature, which translated into English means 'write protection'. Having an unprotected bootloader implies that it can be overwritten by accident - that is a trade-off between performance and reliability.

With this approach, main application requires a few adjustments. By default, SDCC will strip any unused interrupt handlers, which is not what we want. The easiest way to force SDCC to populate the whole IVT is to declare an empty interrupt handler for the last ISR:

```
1 void isr29() __interrupt(29) __naked {; }
```

Finally, the size of the IVT must be subtracted from the application address, so if we compiled with `--code-loc 0x8400` option before, we'll have to use `0x8380` instead.

## Squeezing the last bytes

After a few optimizations the bootloader size was slightly below 700 bytes. That still wasn't good enough for me, since I was aiming at less than 640 bytes (10 blocks). The obvious hot-spot was the IVT: if we relocate it by redirecting interrupt handlers we waste space and if we overwrite it by the bootloader we introduce some additional code, thus still wasting some space.

Ideally, I wanted to implement my own interrupt table, however, it seems that it's hard-coded inside SDCC. After spending some time with the documentation and browsing through the mailing lists, I just ended up looking at the compiler's source code. There is a function

`createInterruptVect()` inside `SDCCglue.c` which is responsible for generating the interrupt vectors. As it turns out, it checks whether or not `main()` is implemented and then proceeds with the interrupt table generation. So the solution was pretty simple: rename `main` into `bootloader_main` and no interrupt vectors will be generated.

The initialization code is also omitted in this case, but that's not a big deal - I simply copied the default initialization and added it to my interrupt table implementation:

```
1  .module INIT
2  .macro jump addr
3      jp 0x8400 + addr
4      .ds 1
5  .endm
6
7  .area IVT
8  int init ; reset
9  jump 0x4 ; trap
10 jump 0x8 ; int0
11 jump 0xc ; int1
12 ; ... ; int2..28
13 jump 0x7c ; int29
14
15 .area GSINIT
16 init:
17     ldw x, #l_DATA
18     jreq 00002$
19 00001$:
20     clr (s_DATA - 1, x)
21     decw x
22     jrne 00001$
23 00002$:
```



```
24     ldw x, #l_INITIALIZER
25     jreq 00004$
26 00003$:
27     ld a, (s_INITIALIZER - 1, x)
28     ld (s_INITIALIZED - 1, x), a
29     decw x
30     jrne 00003$
31 00004$:
32     jp _bootloader_main
```

I created a macro for relocating interrupt vectors, so that it would be easier to specify boot address if it needs to be changed. In this case `jp` instruction is used instead of `int` so the overhead is just 1 CPU cycle (there's also pipeline stall, but that's a whole different topic). One padding byte has to be added due to `jp` using a 16-bit address instead of 24-bit.

Now we can assemble the code with the `-g` option, which tells the assembler to treat all undefined symbols as external - they will be resolved by the linker afterwards.

```
1 sdasstm8 -log init.s
```

We also need to pass these two options to the linker: `-wl-bIVT=0x8000 -wl-bGSINIT=0x8080`. This tells the linker to place IVT section at the beginning followed by GSINIT and rest of the code.

Eventually, the bootloader ended up occupying around 550 bytes, which I could squeeze down to 500 if I stripped unused interrupt vectors and removed initialization code. And of course the bootloader no longer has to stay unsecured.

## Benchmarking

Let's see whether the upload speed is any good. First, let's upload an empty 6k binary via SWIM with `stm8flash`:

```
1 $ time stm8flash -c stlinkv2 -p stm8s003f3 -w empty.bin
2 Determine FLASH area
3 Writing binary file 6144 bytes at 0x8000... OK
4 Bytes written: 6144
5
6 real    0m2.789s
7 user    0m0.000s
8 sys     0m0.024s
```

Now let's repeat the same test with the bootloader:

```
1  $ time python boot.py empty.bin
2  Need to send 96 chunks
3  64
4  128
5  192
6  ...
7  6144
8  Done
9
10 real    0m1.596s
11 user    0m0.048s
12 sys     0m0.008s
```



Not bad. Initially, I wanted to compare the upload speed against the official STVP programming utility, but I was too lazy to register on ST's website solely for the purpose of downloading this utility. So let's just say that it's good enough.

Overall, I'm quite pleased with the results. Despite SDCC having a few limitations, none of them were show-stopping and the bootloader ended up being reasonably compact and fast.

As always, code is on [github](#).

---

#bootloader #sdcc #stm8 #tutorial

 Comments  Share

## NEWER POSTS

[Power analysis with Rigol DS1000Z](#)

## OLDER POSTS

[Mixing C and assembly on STM8](#)

## 6 Comments

[1 Login](#) ▼

G

Join the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS [?](#)

♡ 1

Share

[Best](#) [Newest](#) [Oldest](#)**Vivekanand Dhakane**

2 years ago

Can I use same for cosmic compiler ?

0 0 Reply • Share ›

**Vivekanand Dhakane**

2 years ago

Hi , Thank you very much for this post

0 0 Reply • Share ›

A

**Анна Макарова**

6 years ago

Long time no see, Lujji,

I knew RAM execution will worth it some day and this day has come with ability to write flash in blocks) Some of bootloader functions even can be used in the main firmware to conserve memory. Great work, thanks for the inspiration and detailed alorgythm of thinking while programming STM8S, it's priceless.

0 0 Reply • Share ›

**Alan**

6 years ago

Hi Lujji,

As usual a nice post! I liked the way you did the for loop to detect the 0xDEADBEEF end of file/communication.

BR, Alan

0 0 Reply • Share ›

L

**lujji** Mod

➔ Alan

6 years ago

Thanks, it usually saves a few bytes when doing it this way as opposed to

arrays.

0 0 Reply • Share ›

**Icraft Crafts**

5 years ago

Hi. I'm just starting with stm8 and using the low power version as a start. I'm really enjoying reading your articles.

## RECENT POSTS

[Power analysis with Rigol DS1000Z](#)

[Serial bootloader for STM8](#)

[Mixing C and assembly on STM8](#)

[Executing code from RAM on STM8](#)

[Bare metal programming: STM8 \(Part 2\)](#)

[Bare metal programming: STM8](#)

[Installing Black Magic via ST-Link bootloader](#)

[HTTP server with WebSockets on ESP8266](#)

[Reverse-engineering ST-Link firmware - Part 2](#)

[Reverse-engineering ST-Link firmware](#)

[Adding Trace support to ST-Link clones](#)

© 2023 Iujji

Iujji at protonmail com