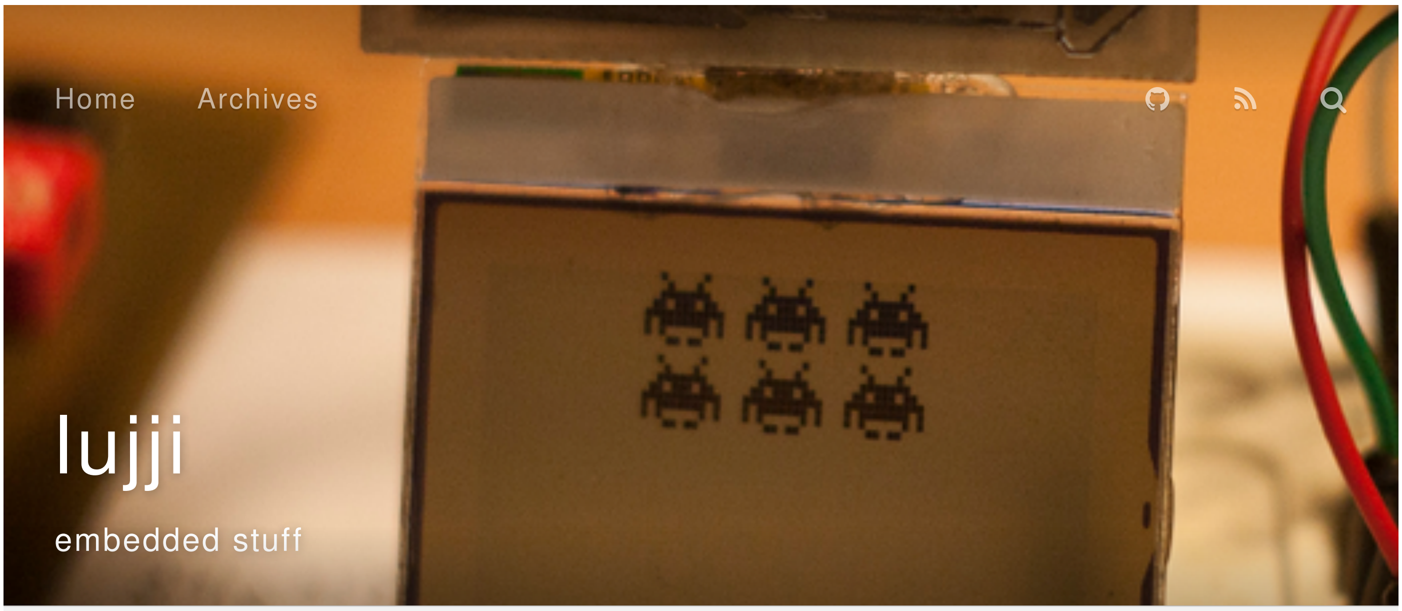


2017-04-11

Bare metal programming: STM8 (Part 2)

In this part we are going to focus on more features of STM8 (clock, EEPROM, option bytes, flash access) and stick some wires into the mains outlet.

Contents:

Clock

STM8 can run on one of 3 different clock sources:

- External clock/crystal oscillator (HSE)
- Internal 16 MHz RC oscillator (HSI)
- Internal 128 khz RC oscillator (LSI)

These clock sources determine the frequency of master clock which clocks the CPU and peripherals. HSI clock can be scaled down by adjusting 2-bit `HSIDIV` prescaler. At startup the master clock source is automatically selected as HSI / 8, which results in 2 MHz. It is possible to decrease CPU frequency by increasing the prescaler value in `CPUDIV` register. By default the prescaler is set to 1.

In the previous part we didn't bother configuring clocks and therefore were running at 2 MHz. This time we'll be using an external crystal connected between PA1 and PA2 pins. Before we start configuring clocks, let's take advantage of processor's clock output capability - this will allow us to perform a sanity check and see if we configured things properly.

```
#define F_LSI_CCO       0x02
#define F_HSE_CCO       0x04
#define F_CPU_CCO       0x08
#define F_HSI_CCO       0x22

void clk_out_enable() {
    /* Configure PC4 as output */
    PC_DDR |= (1 << 4);
    /* Push-pull mode, 10MHz output speed */
    PC_CR1 |= (1 << 4);
    PC_CR2 |= (1 << 4);
    /* Clock output on PC4 */
    CLK_CCOR |= (1 << CLK_CCOR_CCOEN) | F_CPU_CCO;
}
```

Various clock output options are available in the `CLK_CCOR` register - I defined some of the possible ones so that you get the idea.

Enabling external oscillator is done by setting `HSEEN` bit in `CLK_ECKR` register. As soon as the oscillator is ready (which is indicated by `HSERDY` bit), we need to switch the master clock to HSE by writing 0xB4 into `CLK_SWR`. Finally, we wait until the clock source is stabilized and execute the clock switch by setting `SWEN` bit in `CLK_SWCR`.

```
void hse_enable() {
    /* Enable HSE crystal oscillator */
    CLK_ECKR |= (1 << CLK_ECKR_HSEEN);
    while (!(CLK_ECKR & (1 << CLK_ECKR_HSERDY)));

    /* Switch master clock to HSE */
    CLK_SWR = 0xB4;
    while (!(CLK_SWCR & (1 << CLK_SWCR_SWIF)));
    CLK_SWCR |= (1 << CLK_SWCR_SWEN);
}
```

External clock

There is a nice trick when you need to sync two or more microcontrollers: instead of using a crystal you can supply an external clock to the oscillator input pin and leave the other pin floating. STM8 even has a dedicated mode for external clock source, which can be activated by enabling `EXTCLK` option bit.

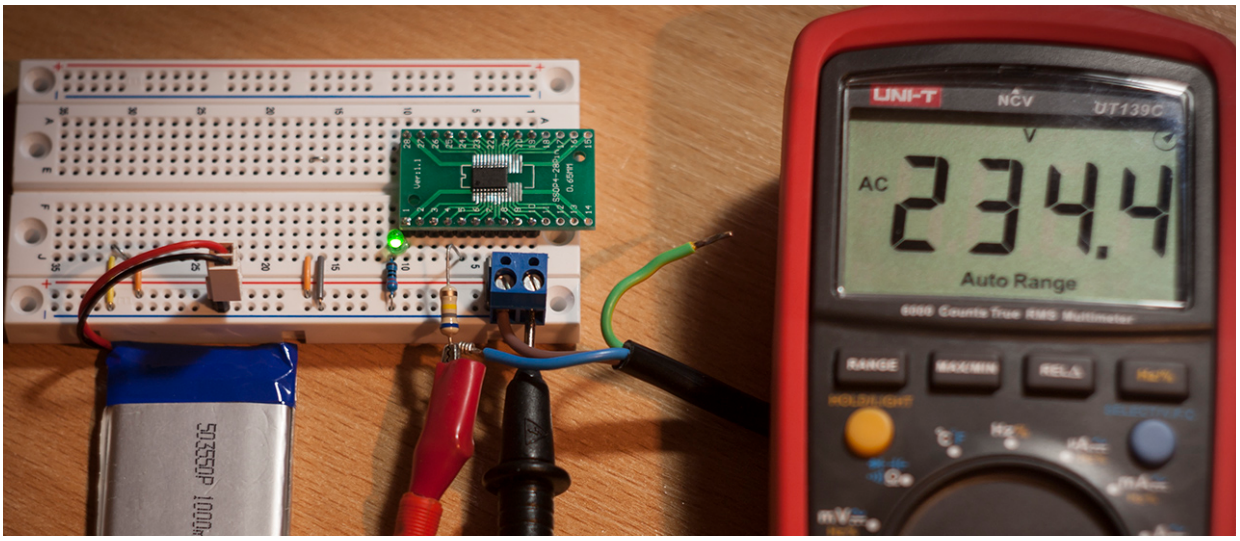
The procedure for enabling external clock is identical to enabling HSE, except that we don't write `HSEEN` bit, since we're not driving an oscillator. In this case I used automatic clock switching mechanism: the only difference is that it allows the processor to run and execute instructions while the clock is being stabilized, although I'm still polling for `SWIF` since I want to stall the CPU until the clock switching is complete.

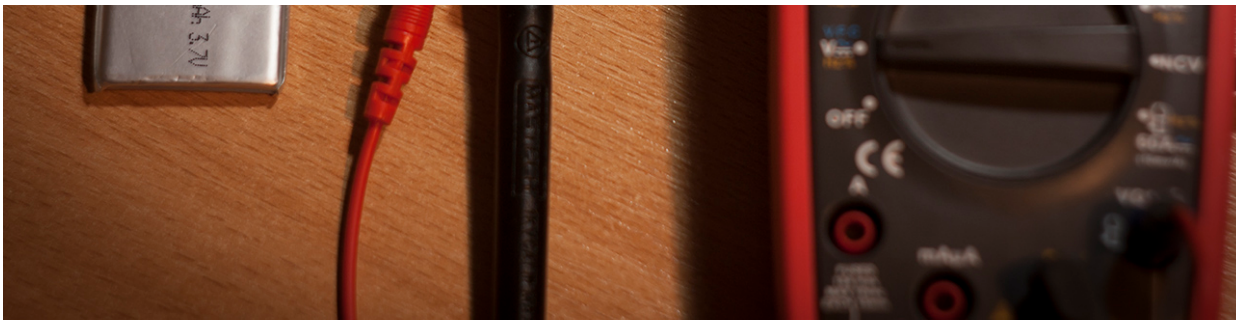
```c
void external_clock_enable() {
    /* set prescaler to 1 */
    CLK_CKDIVR = 0;

    /* Switch master clock to HSE */
    CLK_SWCR |= (1 << CLK_SWCR_SWEN);
    CLK_SWR = 0xB4;
    while (!(CLK_SWCR & (1 << CLK_SWCR_SWIF)));
}
```

After connecting the external clock source to OSCIN (PA1), we need to enable `EXTCLK` option bit by writing 0x08 into `OPT4` (option bytes will be discussed later on). Finally, we call `external_clock_enable()` and wait until the CPU switches clocks. We can ensure that clock switching is successful by enabling clock output and probing CLK_CCO pin.

The external clock source has to be a square wave with 50% duty cycle. According to the documentation, sine and triangle waveforms can be used as clock sources as well. The datasheet also claims that minimum CPU frequency is 0 Hz.

So, presumably, it's possible to clock the CPU from a sinusoidal signal with frequency all the way down to DC. Ugh.. the temptation is irresistible. I don't have a signal generator. *But I do have mains AC.*




What could possibly go wrong?

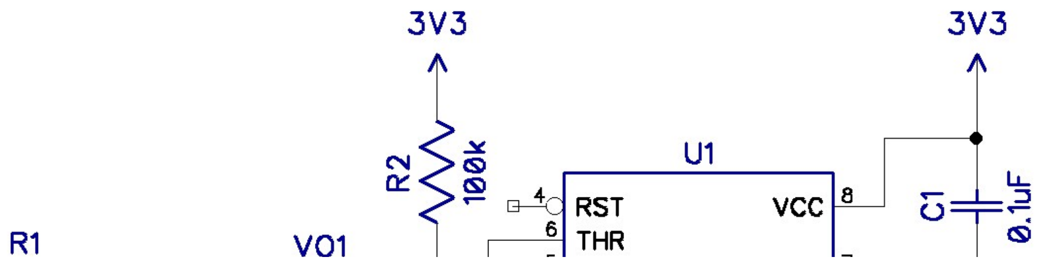
Surprisingly, it worked. In case you're wondering, the chip was able to survive 230V applied to it due to the fact that every pin on STM8 (except for 'true open-drain' pins) has protection diodes to Vcc and ground. These diodes will clamp any excessive voltages and prevent the part from releasing the magic smoke. Having a large value series resistor limits the current flowing through these diodes. The documentation doesn't specify maximum current for clamping diodes, but it's usually a good idea to keep it below 1mA. Keep in mind that when the diodes conduct the current has to return *somewhere* - in this case it's the lithium battery. Did I mention that it's not a particularly good idea?

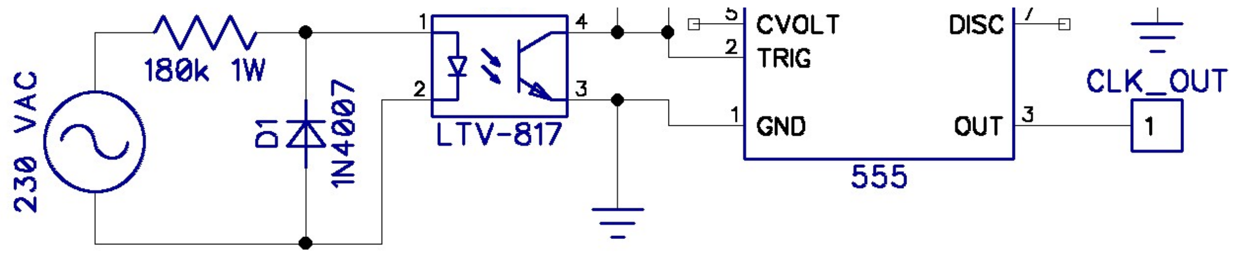
A less lethal approach

Although clocking the MCU directly from mains was kind of fun, I felt rather uncomfortable working when the microcontroller is live. Since I still wanted to find out how useful a mains-clocked processor is, I decided to make things a bit less hazardous by adding some opto-isolation.

I discovered that the clock input circuitry does not like low frequency signals with slow edges. Feeding the output of the opto-isolator directly into the microcontroller results in random glitches on the clock output due to false triggering, and occasionally the CPU just locks up. Applying mains directly worked better since the slew rate was higher in that case. The datasheet hints that HSE has to be above 1 MHz for external crystal, but doesn't specify the lower limit for external user clock. I'm pretty sure we can get stable operation if we improve transition speed. For this purpose we'll need to use a Schmitt trigger, which is basically a comparator with hysteresis, to convert the output of the optocoupler into a nice and clean square wave.

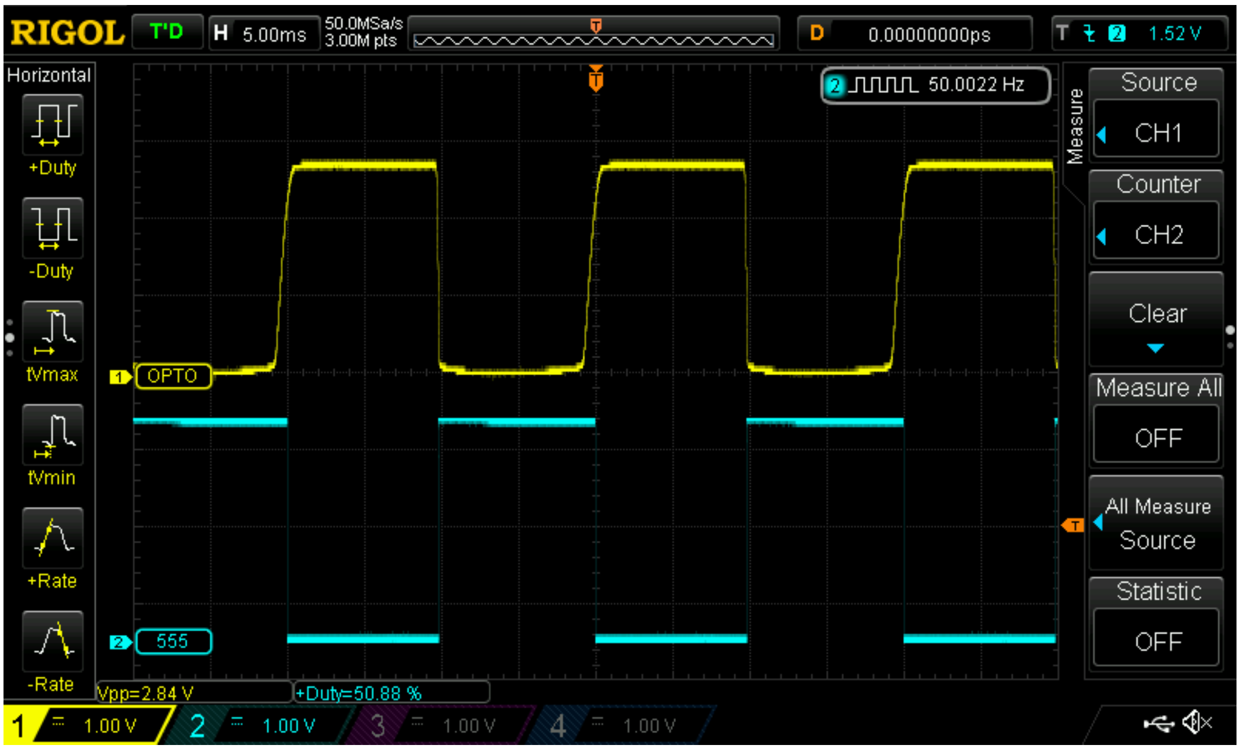
There are dedicated Schmitt trigger ICs and even opto-isolators with Schmitt trigger outputs - non of these do I have at hand. One can implement a Schmitt trigger using an op-amp, but if you want to save a few resistors you can use a good old 555 timer. The 555 has two comparators configured to fire off when voltage on their inputs reaches 1/3 and 2/3 Vcc respectively, which is just about right to set the hysteresis. Comparator inputs are pins 2 (Trigger) and 6 (Threshold). Below is the resulting schematic.





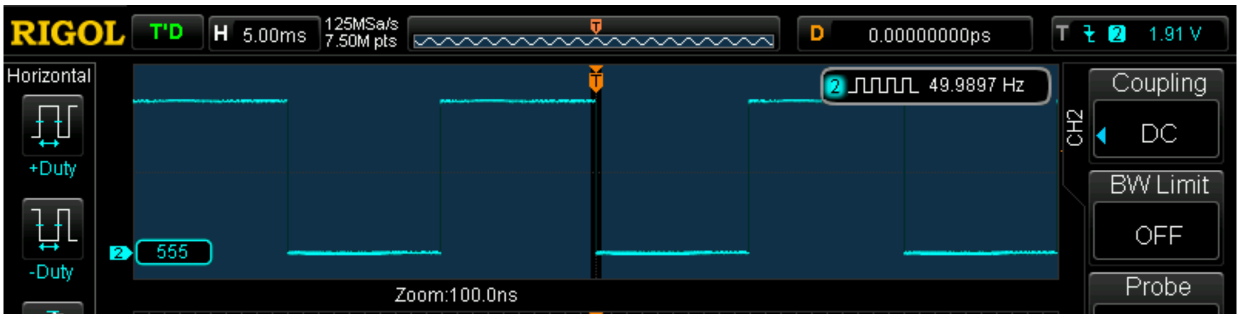
R1 is limiting the current flowing through the LED inside the optocoupler and D1 guarantees that reverse breakdown voltage of the LED would not be exceeded during the negative half cycle of the sine wave. Resistor values depend on the optocoupler being used. In my case I used LTV-817 (Vf = 1.2V) and R1 will limit the peak current to Ipeak = (325 - 1.2) / 180000 = 1.8mA. Since the threshold is fixed, I had to adjust R2 to get as close to 50% duty cycle as possible, which is why it’s value ended up being higher than it should be.

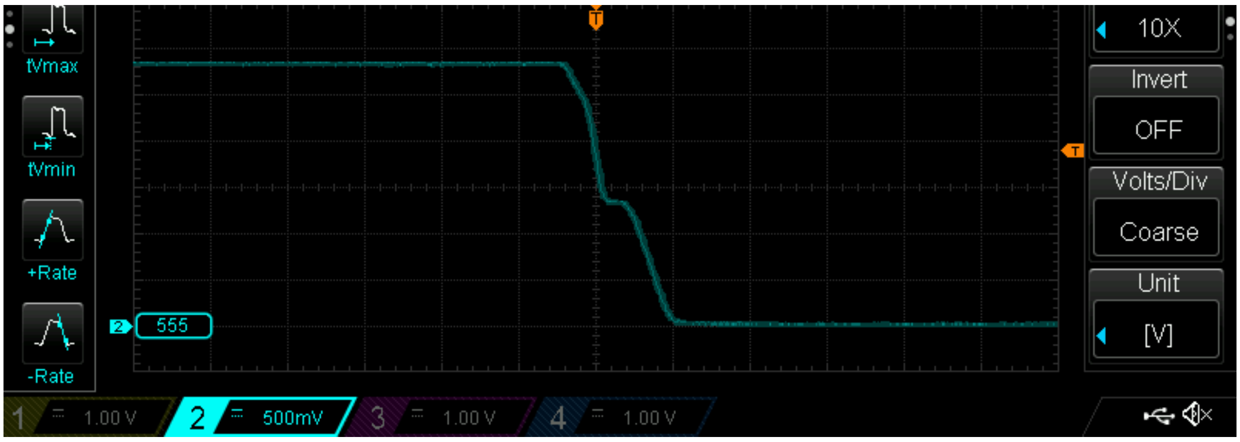
The only timer that I had was NE555 - it’s quite a slow chip not rated for 3.3V operation. A CMOS timer like LMC555 would be a much better choice. That being said, the resulting waveform is still acceptable.



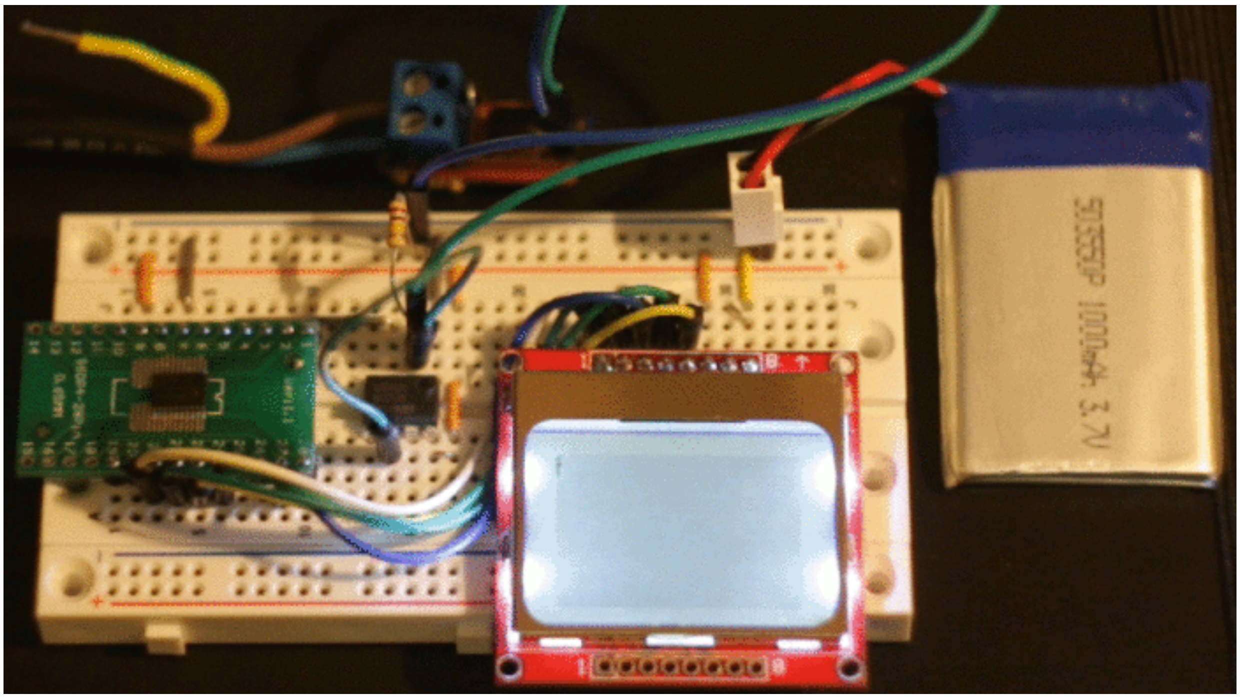
Schmitt Trigger

The duty cycle isn’t precisely 50% and the edges are a bit jagged and jittery, but still good enough for the processor to latch onto.





After connecting the output of the Schmitt trigger to OSCIN, I ensured that I was getting a stable 50 Hz output on CLK_CCO pin and tried bringing up various peripherals. Below is the sped up footage of one of my experiments.


That took about 3 minutes..

Well, that was the slowest SPI communication I've ever seen. Nevertheless, it was nice to know that the processor is still usable at such low clock frequencies.

EEPROM

EEPROM is a small area of memory that can be used for storing things like configuration, calibration data, etc. On STM8S003 EEPROM starts at address 0x4000 and ends at 0x407F, which results in stunning 128 bytes of data. Let's define some macros first. We'll use the first macro for memory access, just like we did with register definitions in part 1.

```
#define _MEM_(mem_addr)         (*(volatile uint8_t *)(mem_addr))

#define EEPROM_START_ADDR       0x4000
```

```
#define EEPROM_END_ADDR         0x407F
```

By default, EEPROM is write protected and a specific sequence is required in order to unlock it: two hardware keys have to be written into `FLASH_DUKR` register. The first time I tried programming EEPROM it didn't work. The reason was me ignoring the following statement in the reference manual: *"before starting programming, the application must verify that the DATA area is not write protected"*. I interpreted it as "you shouldn't write into write-protected areas" while the real meaning was "it takes some time to unlock EEPROM".

```
void eeprom_write(uint16_t addr, uint8_t *buf, uint16_t len) {
    /* unlock EEPROM */
    FLASH_DUKR = FLASH_DUKR_KEY1;
    FLASH_DUKR = FLASH_DUKR_KEY2;
    while (!(FLASH_IAPSR & (1 << FLASH_IAPSR_DUL)));

    /* write data from buffer */
    for (uint16_t i = 0; i < len; i++, addr++) {
        _MEM_(addr) = buf[i];
        while (!(FLASH_IAPSR & (1 << FLASH_IAPSR_EOP)));
    }

    /* lock EEPROM */
    FLASH_IAPSR &= ~(1 << FLASH_IAPSR_DUL);
}
```

Note that on low density STM8S microcontrollers the CPU is stalled during EEPROM write operation, therefore it is not necessary to poll for `EOP` flag.

Reading EEPROM is achieved the same way you read any other memory:

```
void eeprom_read(uint16_t addr, uint8_t *buf, int len) {
    /* read EEPROM data into buffer */
    for (int i = 0; i < len; i++, addr++)
        buf[i] = _MEM_(addr);
}
```

Interestingly, flash programming manual states that on low density devices, EEPROM is comprised of additional 640 bytes of memory located in the same memory array with flash. In other words, it seems like there are 10 pages of flash memory reserved for EEPROM. Also, the manual gives the exact value (it doesn't say *up to* 640 bytes), which contradicts with the datasheet.

Let's try shifting `EEPROM_END_ADDR` to 0x4280 and filling the whole range with dummy bytes:

```
void main() {
    FLASH_DUKR = FLASH_DUKR_KEY1;
    FLASH_DUKR = FLASH_DUKR_KEY2;
    while (!(FLASH_IAPSR & (1 << FLASH_IAPSR_DUL)));

    for (uint16_t addr = EEPROM_START_ADDR; addr < EEPROM_END_ADDR; addr
        _MEM_(addr) = 0xAA;

    while (1);
}
```

Now we can dump EEPROM and check if it was written. I deliberately specified `stm8s103f3` to read more memory than our part has.

```
stm8flash -c stlinkv2 -p stm8s103f3 -s eeprom -r dump.bin
```

Yeap, it worked on every processor that I tried. Although I would rather prefer having a bit more flash memory, it's still good to know that STM8S003 has some extra EEPROM.

Option bytes

Option bytes are located in the EEPROM and allow configuring device hardware features such as readout protection and alternate function mapping. Each option byte, except for read-out protection, has to be stored in a normal form (OPTx) and complementary form (NOPTx). The procedure for writing option bytes is the same as for writing EEPROM, except for the unlcok sequence: `OPT` bit has to be set in `FLASH_CR2` and `FLASH_NCR2` registers.

```
void opt_write() {
    /* new value for OPT5 (default is 0x00) */
    uint8_t opt5 = 0xb4;

    /* unlock EEPROM */
    FLASH_DUKR = FLASH_DUKR_KEY1;
    FLASH_DUKR = FLASH_DUKR_KEY2;
    while (!(FLASH_IAPSR & (1 << FLASH_IAPSR_DUL)));

    /* unlock option bytes */
    FLASH_CR2 |= (1 << FLASH_CR2_OPT);
    FLASH_NCR2 &= ~(1 << FLASH_NCR2_NOPT);

    /* write option byte and it's complement */
    OPT5 = opt5;
    NOPT5 = ~opt5;

    /* wait until programming is finished */
    while (!(FLASH_IAPSR & (1 << FLASH_IAPSR_EOP)));

    /* lock EEPROM */
    FLASH_IAPSR &= ~(1 << FLASH_IAPSR_DUL);
}
```

If you mess things up, you can reset the option bytes via SWIM:

```
$ echo -ne '\x00\x00\xff\x00\xff\x00\xff\x00\xff\x00\xff' > opt.bin
$ stm8flash -c stlinkv2 -p stm8s003f3 -s opt -w opt.bin
```

Interestingly, if we read the memory a bit further, we find a section which contains the following data:

```
0x480B: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0
0x4823: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0
0x483B: 00 00 00 00 00 0c f3 12 ed 12 ed cd 32 77 88 49 b6 01 fe 20 df 0
0x4853: fe 00 00 00 00 00 00 00 00 00 00 00 00 57 00 1f 5b 00 00 1e 00 3
0x486B: 36 31 34 32 31 33 1f 00 00 1f 00 00 00 00 00 00 00 00 00 00 00
```

20 bytes at address 0x4840 are written with their complement values just like the option bytes. This whole block is write protected and differs slightly from one processor to another - unique ID perhaps?

Flash

One thing that I like the most about STM8 is flash access.

The two most common types of flash memory are NAND and NOR flash. Flash is physically divided into blocks, which may be further divided into sectors. The entire memory is linear and can be read or written in a random access fashion, however both NAND and NOR flash share the same disadvantage: you can flip a 1 into a 0 but not vice-versa. The only way to flip a 0 back to 1 is to erase the whole block. If you need to overwrite a few bytes in flash memory you have to buffer the whole page into RAM, modify the buffer, erase flash page and write the buffer back into flash memory - the whole process is rather time-consuming.

With STM8 this is not the case: the whole memory can be accessed at byte level. You can write any byte inside any page and erase it by simply writing 0x00 at that address. Essentially, you can treat flash memory as a large EEPROM.

Removing write protection is almost identical to unprotecting EEPROM.

```
void flash_write(uint16_t addr, uint8_t *buf, uint16_t len) {
    /* unlock flash */
    FLASH_PUKR = FLASH_PUKR_KEY1;
    FLASH_PUKR = FLASH_PUKR_KEY2;
    while (!(FLASH_IAPSR & (1 << FLASH_IAPSR_PUL)));

    /* write data from buffer */
    for (uint16_t i = 0; i < len; i++, addr++) {
        _MEM_(addr) = buf[i];
        while (!(FLASH_IAPSR & (1 << FLASH_IAPSR_EOP)));
    }

    /* lock flash */
    FLASH_IAPSR &= ~(1 << FLASH_IAPSR_PUL);
}
```

Just like with EEPROM, we can dump the entire flash memory:

```
stm8flash -c stlinkv2 -p stm8s003f3 -s flash -r dump.bin
```

SDCC has various attributes like `__xdata` and `__eeprom` for placing things in specific memory locations. Unfortunately, none of them are implemented for STM8 yet. We can partially work around this limitation by using `__at` attribute:

```
/* Use last 64 bytes of flash for user data */
#define ID_ADDR                 (0x8000 + 0x1FC0)
#define USER_DATA_ADDR          (ID_ADDR + 1)

/* Tell compiler where the variables are located */
__at(USER_DATA_ADDR) uint8_t data[8];
__at(ID_ADDR) const uint8_t id = 42;
```

Let's take a closer look at the above example: first we define two addresses in flash memory. Remember that program memory starts at 0x8000, so we add this value to get the address we want. Next we declare `data` array with attribute `__at(USER_DATA_ADDR)` - this tells the compiler where to look when the variable is being accessed. For example, a read operation on `data[2]` will return the value at address 0x9FC3, which is the same as calling `_MEM_(0x9FC3)`. Same goes for write operation: if flash is unlocked, then writing `data[2]` will store the value at the appropriate address in flash memory. If flash unlock sequence was not executed before performing a write, then `WR_PG_DIS` bit will be set in `FLASH_IAPSR` register to indicate an attempt to modify write-protected page.

The second variable `id` is declared as `const` - this will actually produce a binary with the value placed at specified memory address. Declaring a variable as constant means that compiler will not allow us to perform explicit write operations, unless we write directly at the specified address (which kind of defeats the purpose of `const` qualifier). Unfortunately, this approach will not work for EEPROM - SDCC will simply produce a larger binary image.

That's it for now. In the next part we're going to take a look at some of the features essential for real-world applications and discuss questions of reliability and performance. As always, code is available on github. Since previous article the repository evolved into a small peripheral library with dedicated examples directory.

#bare-metal #sdcc #stm8

Comments Share

NEWER POSTS

Executing code from RAM on STM8

OLDER POSTS

Bare metal programming: STM8

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

Recommend 1 Tweet Share Sort by Best

Join the discussion…

LOG IN WITH OR SIGN UP WITH DISQUS ?

Name

Анна Макарова • 4 years ago

Dear Lujji, Thanks a lot, I've been monitoring your blog for a while being afraid I was too greedy to ask you so much before, and I'm glad you found the time to teach us more, hope we do not bother you and you are as interested in the subject as we are =^_^=

2 | • Reply • Share ›

Joe • 5 months ago • edited

Sdd ver 3.5 is available with apt-get in Raspbian. Where would the stm8flash go? Apt installed sdcc in /usr/bin and sdcc is an executeable, not a directory.. Could we use ~/local/sdcc to hold our code and the stmflash anyway? Stm8flash won't compile on my Raspbian Stretch. Found there is a windows version and it's files are compatable with STVP programmer.

| • Reply • Share ›

Daniel Beneš • 2 years ago

Hello.
Thanks for the guide, but for some reason, I'm still not able to get it work properly. I always end with Error in complementary bytes (when I try to rad out back the option bytes in STVP)
I use SDCC, I checked what values are writen when I use graphic STVP and tried to set exactly the same.
I'm working with STM8s005k6, the setting is called in setup which is the first thing in main loop.
My code:

```
// Set option bytes because of buzzer
if(OPT->OPT2 != 0x80 || OPT->OPT3 != 0x08){
FLASH->CR2 |= FLASH_CR2_OPT; //unlock option bytes for writing
FLASH->NCR2 &= (uint8_t)(~FLASH_NCR2_NOPT);
while (!(FLASH->IAPSR & FLASH_IAPSR_DUL));

OPT->OPT2 = 0x80; // enable LSI clock source
OPT->NOPT2 = 0x7f;
OPT->OPT3 = 0x08; // set PD4 as alternative buzzer output
OPT->NOPT3 = 0xf7;
while (!(FLASH->IAPSR & FLASH_IAPSR_EOP)); // wait for write finish
```

Thank you.

Reply • Share ›

tovis • 3 years ago

Hi! The strange behavior of TIM4 isr was caused by low supply voltage. I have used 3,3V output of STLINK-V2 got from eBay, but in real it was about 2,7V what is not enoguh for 16MHz cpu clock. I have changed to external supply (trough micro USB) and I have got expected result using PD_ODR ^= (1 << 3)
Insufficient optimization problem still exist, especially for bit manipulation :(

Reply • Share ›

tovis → tovis • 3 years ago

That time occasionally I have used SDCC 3.6.0! Later I've got 3.6.9 it's doing better.

Reply • Share ›

lujji Mod → tovis • 3 years ago

Hi, glad you solved it. I actually covered code optimization in my bootloader article - some changes were made to SDCC since then, so bit manipulations are faster now. Also, you can check out my git repo for some additional optimization rules: https://github.com/lujji/st...

Reply • Share ›

tovis → lujji • 3 years ago

I'm glade to hear that you do not abandon this project. Now I'm stuck on UART - something wrong I'm sending 0xAA (usual test byte) but receive 0x80? I'm suspect level problems around MAX3232 used as a driver.
I will checkout boot loader later - I need to get working every peripherals.
Your work really helpful!

Reply • Share ›

tovis • 3 years ago • edited

Well done! Thanks you that write it down!
I'm just now started to
explore this mcu and tools such as SDCC, but I quickly found some problems. I have tried to build your/mine interrupt driven (1msec) tick counter using TIM4. Use the same technic as you toggle port bit/pin and put this code to isr:
PD_ODR ^= 0B00001000;
TIM4_SR &= 0B111111110;
(To compile this you need to put --std-sdcc99 option for sdcc)

• Reply • Share ›

ter_min4 • 4 years ago

Hello Lujji,
I see you used stm8flash
Do you happen to know how to sniff USB, so we can also start to flash more than 32kB?
Maybe sniff USB while sending limited range like 0x010000 - 0x010010 with ST Visual Progammer (Windows)
Somebody posted issue:
https://github.com/vdudouyt...
Thanks

• Reply • Share ›

lujji Mod ter_min4 • 4 years ago

Wireshark is the best tool for the job if you want to sniff USB. Unfortunately, I don't have any 64k parts on hand so I can't help much.

• Reply • Share ›

Анна Макарова • 4 years ago • edited

I've encountered http://www.st.com/en/develo...
Runs on my Linux Mint very well. Unfortunatelly, does not really generate code templates with HW initializations (yet?). I''m reading their manual just now. Power calculator section in the app hints that power consumption is different for the code being executed from RAM compared to FLASH. Speed can be different also. Have you tries to place your code in RAM and gather some real experience?

• Reply • Share ›

Анна Макарова Анна Макарова • 4 years ago

Apparently, on some STM8's people copy some code to RAM then jump there turning FLASH memory off completely somehow. Together with low power oscillator it allows them to conserve lots of power while not sleeping entirely. I wonder if it's possible on the cheapest one we have. Also there is FLASH overclocking limitation that could be potentially overridden with RAM fetching. I wonder how many people have tried to squeeze as much as possible from STM8003 in terms mentioned above (starting from extreme power saving while working from RAM, till extreme overclocking yet with the

as sample frequency source to trim internal HSI generator compensation voltage and temperature determined drift. Actually, I'm interested in realtime HSI frequency calibration while emulating some high-speed protocols\signals where clock accuracy is essential. This principleis used in various vUSB implementations even in crystal-less mass-produced USB devices.
ST recommends calibration examples from "firmware library", but last mention of it was 6 years ago and I can not find it after their site rebuilt.

Has anyone tried to play with HSI trimming in realtime? The calibration range seems to be not too wide (-4..+3) and I wonder if it is enough and are there any glitches during runtime.

Talking about internal memories of STM8, I wonder what are their top frequencies (while overclocking), especially while working on low-voltage batteries...

• Reply • Share ›

lujji Mod Анна Макарова • 4 years ago • edited

I haven't read this appnote before and was a bit surprised that ST used mains as frequency reference.
Adjusting HSI trimming value on the fly is possible, the transition is glitch-free. Actually, the first thing I thought about was frequency modulation - too bad there's no internal PLL.
edit: your reply was blocked by disqus for some reason - fixed.

1 • Reply • Share ›

Анна Макарова lujji • 4 years ago • edited

Absence of PLL disappoints me too. Some protocols do need basic frequencies different from even harmonics of 16Mhz (slow USB needs multiples of 12 Mhz, slow Ethernet - 10Mhz, etc.). Being unable to use external crystal we'll stuck with indirect target frequency reconstruction based on received signals from it and real-time phase\frequency compensation of our own clock (basically it's a perverted kind of software-emulated PLL). As far as I understand,

ARCHIVES

RECENT POSTS