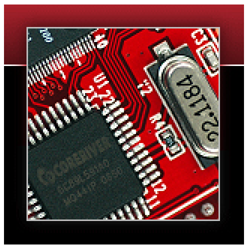


8L-Course, Part 4 - Timing

STM8

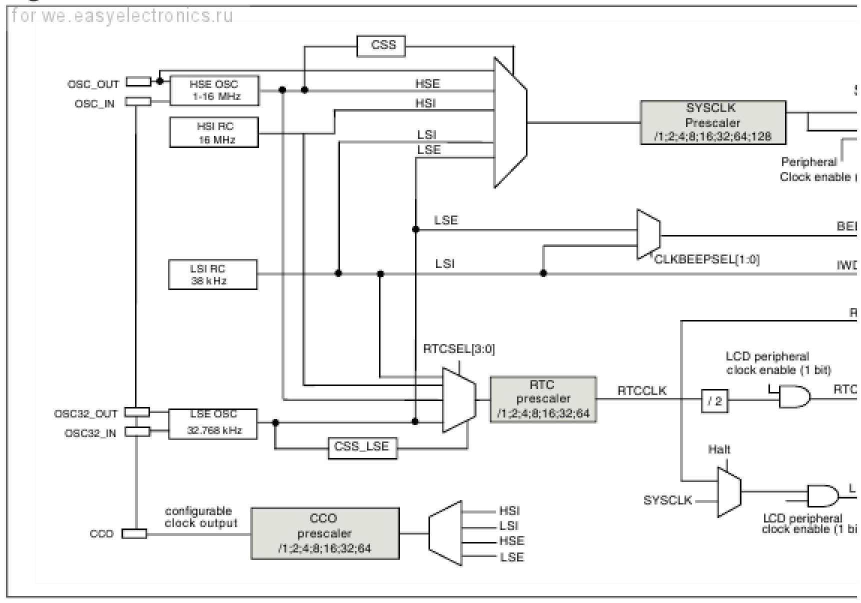
PCBWay
HIGH-QUALITY PCB
ONLY $5 FOR 10 PIECES
• Rogers, HDI, aluminum and rigid-flex PCB are available now
• Production time 24 hours
PCB ASS
Free shipping
ONLY
Compon
Quality a

← Part 3 — Interrupts Contents Part 5 — Timers, the beginning →

Previously, we somehow did not think about what frequency the MCU operates at and what it is clocked from — it works and works well. Delays were chosen by eye, without knowing the clock frequency. It's time to put an end to this and figure out how clocking is organized in STM8.

Here's a scary picture to start with:

Figure 17. Clock structure


for we.easyelectronics.ru
CSS
OSC_OUT
OSC_IN
HSE OSC 1-16 MHz
HSI RC 16 MHz
HSE
HSI
LSI
LSE
SYSCLK Prescaler /1;2;4;8;16;32;64;128
Peripheral Clock enable
LSE
BEE
CLKBEEPSEL[1:0]
LSI RC 38 kHz
LSI
IWD
RTCSEL[3:0]
RTC prescaler /1;2;4;8;16;32;64
RTCCLK
/ 2
LCD peripheral clock enable (1 bit)
RTC
OSC32_OUT
OSC32_IN
LSE OSC 32.768 kHz
CSS_LSE
Halt
SYSCLK
LCD peripheral clock enable (1 bi
configurable clock output
CCO prescaler /1;2;4;8;16;32;64
HSI
LSI
HSE
LSE
CCO

STM8 has 4 clock sources (on the left in the picture):

HSI (High Speed Internal) — high-frequency internal **RC** generator. It operates at **16 MHz** and, importantly, is quite **well calibrated** at the factory. Unlike AVR, where the calibration drifts when switching the RC generator frequency (because the parameters of the generator itself change), here everything is fine. The HSI generator always operates at 16 MHz, and the

Live

Comments Publications

penzet → Sprint Layout in OS X 18 → Software for electronics engineer

Vga → EmBitz 6 → Software for electronics engineer

Vga → Rail-to-rail: ideal operational amplifier or a clever marketing ploy? 1 → Theory, measurements and calculations

Flint → A little more about 1-wire + UART 56 → Hardware connection to the computer.

penzet → Cross-platform terminal - SerIO 3.x 25 → Software for electronics engineer

From Gorn → PIP regulator 2 → Algorithms and software solutions

whoim → CC1101, Treatise on tracing 89 → Blog im. Khomin

OlegG → Clock on Bluetooth LE module 8 → Cypress PSoC

Technicum505SU → Nuances of PWM control of a DC motor by a microcontroller 3 → Theory, measurements and calculations

sunjob → DDS synthesizer AD9833 88 → Blog named after grand1987

dihalt → W801, LCD screen and tar spoon 2 → Detail

nictrace → New Arduino-compatible board. 20 → FPGA

sunjob → Connecting the ARM GCC compiler to CLion 1 → Software for electronics engineers

Vga → CRC32: on STM32 as on PC or on PC as on STM32. 58 → STM32

dmitrij999 → Capturing images from a USB camera using STM32 6 → STM32

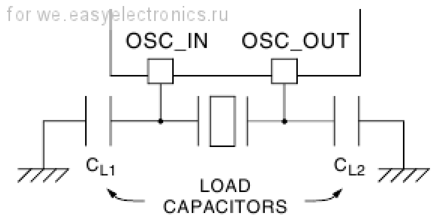
Gilak → Expanding the capabilities of a simple MC up to an ADC on 2 or 1 pin. 8 → Theory, measurements and calculations

sva_omsk → Lithium ECAD - Russian PCB CAD 40 → Software for electronics engineer

x893 → W801 - budget controller with Wi-Fi 4 → Details

podkassetnik → Changing the standard instrument cluster lighting of Logan-like cars 5 → Automotive electronics frequency is reduced, if necessary, by the SYSCLK divider after the generator. At startup, the MC is clocked from the HSI with a divider of 8. That is, the core operates at 2 MHz. Naturally, the factory calibration is not a panacea and the HSI frequency will still drift when the voltage/temperature changes. For this case, an additional calibration is provided. In general, if you need high accuracy and stability of the clock frequency, use ...

HSE (High Speed Extreme) - a high-frequency external generator. Simply put, quartz. **The OSC_IN (A2)** and **OSC_OUT (A3)** pins are used to connect the quartz resonator .



The frequency can be from **1 to 16 MHz** (we ran 20 MHz - nothing, it works).

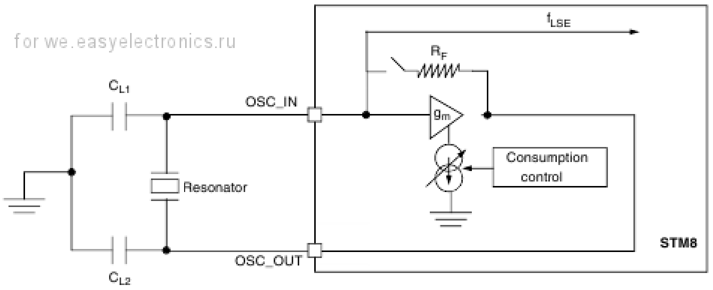
In addition to swinging the quartz resonator, the HSE can work as **an input for an external clock signal** . The OSC_OUT pin is then not used (it can be used as a regular pin), and a signal is sent to OSC_IN.

And also, in case of any troubles with the quartz, there is **an emergency clock switching system - CSS (Clock Security System)** . It monitors the stability of the clock signal from the HSE and if it is suddenly interrupted, it switches the clock to the HSI. Thus, even if the quartz resonator stops ticking, the MC will not freeze, but will continue to work quietly, but on a less stable clock generator.

LSI (Low Speed Internal) is an RC generator with a frequency of about **38 kHz** (the datasheet says something scary: spread from 26 (min) to 56 (max) kHz). This frequency is not calibrated and can fluctuate quite a lot depending on the temperature and supply voltage. It

is used primarily to clock the watchdog timer, but you can also clock the core and RTC from it (of course, they will not be able to count the time accurately, but it will do just fine to wake up the MC by the timer). The STM8L has a very strong energy-saving sorcery "Low power run": in this mode, all generators are disabled except for the LSI (the core is clocked from it), the program is executed from RAM, and the flash memory power is also disabled. Allows you to achieve a consumption of only a few microamps (with the core running!).

LSE (Low Speed External)— a generator for the clock (RTC) and some other peripherals (however, no one forbids clocking the core from it). A "clock" quartz at 32768 Hz is connected to the OSC32_IN (C5) and OSC32_OUT (C6) pins: The



clock signal from it goes through its own RTC Prescaler divider and can then be fed to the LCD controller and the clock (RTC).

Some MC models have their own CSS for the clock generator (we were unlucky, we don't have one). It works on the same principle as the CSS for HSE: if the signal from the clock quartz disappears, the RTC starts to be clocked by the LSI. This, by the way, is a wildly useful thing for devices that spend most of their life in sleep mode, waking up to the "alarm clock" from the RTC. If not for this system, then in case of a clock failure, our device will die without waking up.

Vga → ROPS (Rem Object Pascal Script) - embedded interpreter of the Pascal language. Plugin PSImport_Classes 3 → Algorithms and software solutions

Full broadcast | RSS

1-Wire The other arduino ARM Assembler Atmel AVR C++ compel DIY enc28j60 ethernet FPGA gcc I2C AND KEIL LaunchPad LCD led linux LPCXpresso MSP430 nxp PCB PIC pinboard2 RS-485 RTOS STM32 STM8 STM8L OF UART USB algorithm assembler ADC the library power unit detail display an idea tool competition competition 2 ANGRY microcontrollers for beginners review Debug board soldering iron printed circuit board salary FPGA crafts purchases programmer programming Light-emitting diode software scheme circuit design Technologies smart House photoresist it's free crap Times humor

Blogs

Group

AVR	**38.98**
STM8	**37.92**
Garbage truck	**29.53**
STM32	**28.46**
Detail	**24.63**
Connection of hardware to the computer.	**24.04**
Circuit design	**18.15**
Smart House	**17.75**
MSP430	**17.13**
LPC1xxx	**14.79**

All blogs

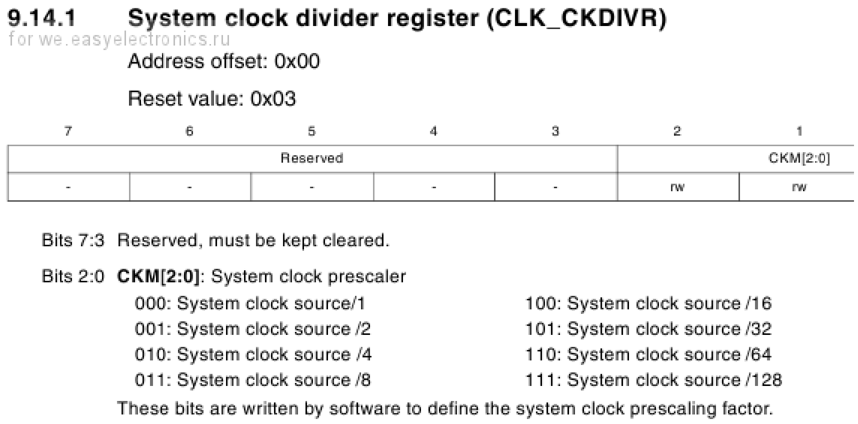
And this way it will be possible to wake up and take some action.

Sometimes (for calibration, for example) it is useful to output a signal from the clock generator to the MCU pin. For this purpose, the STM8 has a gadget called **CCO (Configurable Clock Output)** . It allows you **to output a signal from any clock generator to the CCO (C4) pin** , and even with a configurable prescaler (up to /64).

The system clock signal SYSCLK (see the scary picture at the beginning) goes to the core and all other peripherals (timers, interfaces, ADC). To reduce consumption, in the STM8L, the peripherals are turned off at startup and are not clocked. While there is no clocking, you will not be able to write anything to its registers. Therefore, before starting to work with any peripheral device, you need to enable the clock signal. And in the STM8S, clocking of all peripherals is enabled by default.

So, everything seems clear with the organization of clocking, **but how to use it?**

Let's start with the system clock signal divider (SYSCLK). It is controlled by the **CLK_CKDIVR**



9.14.1 System clock divider register (CLK_CKDIVR)

Address offset: 0x00

Reset value: 0x03

7	6	5	4	3	2	1
Reserved						CKM[2:0]
-	-	-	-	-	rw	rw

Bits 7:3 Reserved, must be kept cleared.

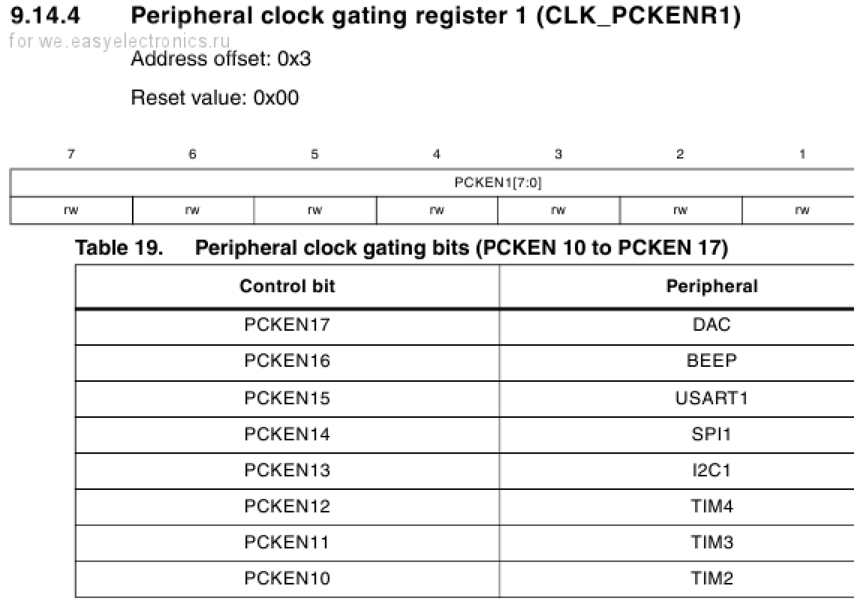
Bits 2:0 **CKM[2:0]**: System clock prescaler

000: System clock source/1	100: System clock source /16
001: System clock source /2	101: System clock source /32
010: System clock source /4	110: System clock source /64
011: System clock source /8	111: System clock source /128

These bits are written by software to define the system clock prescaling factor.

register. Only the three least significant bits are active in it, and the divider can take values **from 1** (i.e. no divider at all) **to 128** (which, with an input frequency of 16 MHz, will give only 125 kHz for core clocking). The divider can be changed at any time and without any additional delays.

In order to supply (or disable) clocking to the desired peripherals, there are **CLK_PCKENRx** registers . There are three of them in the STM8L15x. One bit is allocated for each peripheral device.



9.14.4 Peripheral clock gating register 1 (CLK_PCKENR1)

Address offset: 0x3

Reset value: 0x00

7	6	5	4	3	2	1
PCKEN1[7:0]						
rw	rw	rw	rw	rw	rw	rw

Table 19. Peripheral clock gating bits (PCKEN 10 to PCKEN 17)

Control bit	Peripheral
PCKEN17	DAC
PCKEN16	BEEP
PCKEN15	USART1
PCKEN14	SPI1
PCKEN13	I2C1
PCKEN12	TIM4
PCKEN11	TIM3
PCKEN10	TIM2

A bit set to one enables clocking, and a bit reset to zero disables it.

Now we will not use any peripheral devices, but in the next part we will start playing with timers, and this register will be useful for us there. Therefore, consider that this information was just for general development :)

In AVR, the clock source was set via fuse bits during firmware and it was impossible to change it while the program was running. This is bad form and no one does it now :) **STM8 can change clock sources "on the fly"** , and there are two ways to do this:

Automatic switching of the clock source
First, you need to set the SWEN bit (hello, acoustics lovers! But in fact, this is SWitch ENable) in the CLK_SWCR register. After that, a magic number is entered into the CLK_SWR register:
0x01 - Switch to HSI
0x02 - Switch to LSI
0x04 - Switch to HSE
0x08 - Switch to LSE
(any other magic numbers will be mercilessly ignored)

... the selected clock generator is automatically started, the system waits until it enters the operating mode, and switches the clock. We don't need to worry about anything else.

While the switching has not yet occurred, the SWBSY flag will be raised in the CLK_SWCR register. It can be used to determine the moment when we have switched to a new clock source.

Also, by setting the SWIEN bit (in the same CLK_SWCR), we can catch an interrupt immediately after the switching process is completed. Sometimes it is quite convenient. Here is a handler template:

```
ISR(CLK_interrupt, CLK_SWITCH_vector)
{

 if (CLK_SWCR_bit.SWIF == 1)
 {

  CLK_SWCR_bit.SWIF = 0;
 };
};
```

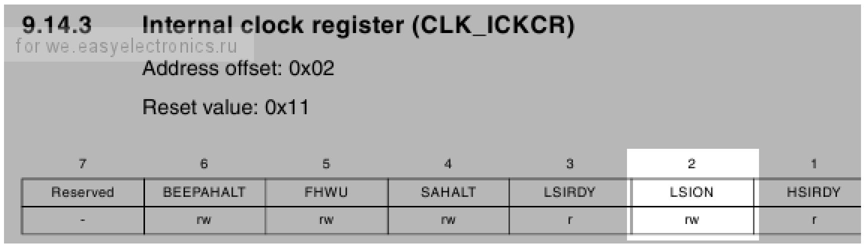
The tricks with if are related to the fact that an interrupt from CSS can arrive to the same vector (if it is enabled, of course) and it is necessary to figure out what happened before processing.

Switching the clock source "manually"
Automatic switching has one drawback - we do not decide when exactly the transition to a new clock source will occur. Of course, in 95% of cases this is not of fundamental importance and you can just wait in the cycle until the SWBSY flag drops, but sometimes you need to switch at a strictly defined moment. To do this, you need to do everything the other way around:

First, enter the number of the desired source in the CLK_SWR register (the numbers are the same as last time). The selected generator starts and goes into operating mode. The SWIF bit in the CLK_SWCR register will tell us that it is ready for operation. Or an interrupt, if enabled. But you cannot rely on the SWBSY bit here - it will be 1 until the process of switching to a new generator is completed.

After the generator has started, we can set the SWEN bit at the right moment and the system will immediately switch to the new clock source.

The old source, if it is used for something else (for example, HSI for SWIM) is not switched off automatically after switching. If you need to switch it off, you can reset the corresponding bit in the CLK_ICKCR register.



9.14.3 Internal clock register (CLK_ICKCR)

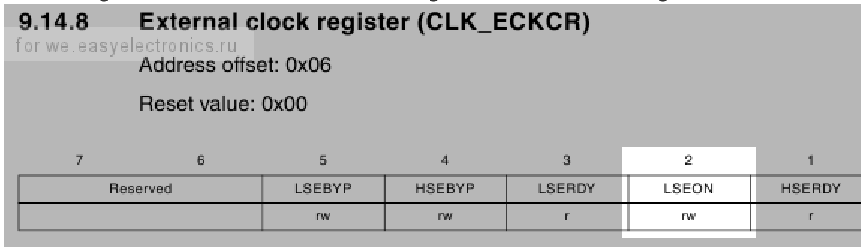
Address offset: 0x02

Reset value: 0x11

7	6	5	4	3	2	1
Reserved	BEEPAHALT	FHWU	SAHALT	LSIRDY	LSION	HSIRDY
-	rw	rw	rw	r	rw	r

The LSION bit is responsible for the low-frequency internal generator, and HSION for the high-frequency one.

External generators are controlled through the CLK_ECKCR register:



9.14.8 External clock register (CLK_ECKCR)

Address offset: 0x06

Reset value: 0x00

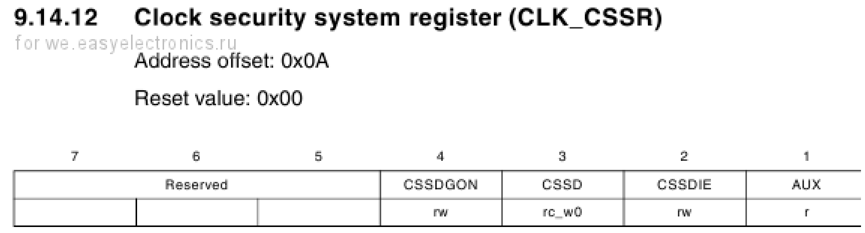
7	6	5	4	3	2	1
Reserved		LSEBYP	HSEBYP	LSERDY	LSEON	HSERDY
		rw	rw	r	rw	r

Here we are interested in the LSEON and HSEON bits.

But a running generator cannot always be switched off by resetting one bit. If the generator is used as an active clock source for the core or some peripherals, then it will not be possible to turn it off. First, you will need to switch the clock to another source.

Okay, let's say we switched the clock to the quartz resonator. **But how can we protect ourselves from trouble now? We seem to have CSS?**

It is controlled by only one register CLK_CSSR,



9.14.12 Clock security system register (CLK_CSSR)

Address offset: 0x0A

Reset value: 0x00

7	6	5	4	3	2	1
Reserved			CSSDGON	CSSD	CSSDIE	AUX
			rw	rc_w0	rw	r

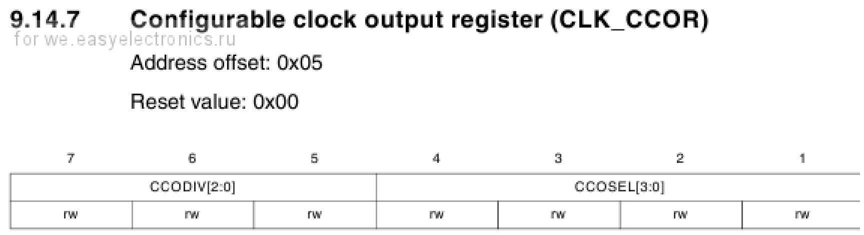
and to enable it, you just need to set a single CSSEN bit. Naturally, we must work from the HSE, otherwise what's the point? By the way, you can turn it on, but not turn it off. CSSEN is reset only when the MCU is rebooted.

After turning on, CSS starts closely monitoring the clock signal from the HSE. If it is suddenly interrupted, the following rescue measures are performed:

- HSI is started (if it is off) **and clocking is switched to it**
- HSE is turned off for good
- The AUX flag is raised in CLK_CSSR , notifying us that the backup generator is working.
- The CSSD flag is also raised there and, **if the CSSDIE bit is set, an interrupt occurs.**
— All registers from the CLK group, except CLK_CKDIVR, are locked for writing. The clock settings (except for the divider) can no longer be changed until the next reboot.

Simply put, the CSS actions in case of a clock failure look like this: TRANSFER CLOCK TO HSI, RAISE FLAGS, LOCK ALL REGISTERS, INTERRUPT AAAA PANIC!!!111 In my opinion, locking the registers is unnecessary. We cannot switch back to quartz (the failure may have been temporary, and the quartz will start normally in a couple of seconds), which means that although the device continues to work, it will not be able to regain full functionality until it reboots.

Well, okay, if in case of a failure we have to work from HSI, it would be a good idea to calibrate it (or at least look at the real frequency). There is a CCO module for outputting a clock signal to the MC pin. It is also controlled by only one register CLK_CCOR:


9.14.7 Configurable clock output register (CLK_CCOR)

Address offset: 0x05

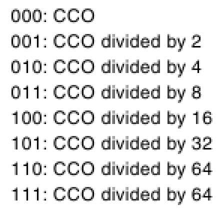
Reset value: 0x00

7	6	5	4	3	2	1
CCODIV[2:0]			CCOSEL[3:0]			
rw	rw	rw	rw	rw	rw	rw

The CCOSEL bits are responsible for selecting the generator, the signal from which will be fed to the pin (C4, let me remind you). The value 0000 corresponds to disabled CCO, when nothing is output to the pin. And the other options are the same magic numbers that we used when switching the clock:

```
0x01 (0001) - Выводим тактовый сигнал от HSI
0x02 (0010) - От LSI
0x04 (0100) - HSE
0x08 (1000) - LSE
```

The CCODIV bits are responsible for the prescaler, through which the signal passes before getting to the pin:


000: CCO
001: CCO divided by 2
010: CCO divided by 4
011: CCO divided by 8
100: CCO divided by 16
101: CCO divided by 32
110: CCO divided by 64
111: CCO divided by 64

The least significant bit CCOSWBSY means that the selected generator is in the process of starting/stabilizing. While it is raised, you cannot write to the other bits.

Having set up CCO, you can take a frequency meter and see how much the frequency of the generator we are interested in deviates from the nominal value. Just do not forget that before catching something on the pin, it must be set to output.

Let's say we measured the HSI frequency and saw that it does not correspond to the nominal value a little. And we need to get exactly 16 MHz. For such cases, the **HSI calibration** option is provided .

In general, as I already said, the HSI in each MCU is calibrated at the factory. The calibration value is stored in the CLK_HSICALR register. Do not ask me in what parrots it is measured.

The user value is written to the CLK_HSITRIMR register. RM0031 strongly recommends that it be in the range from CLK_HSICALR-12 to CLK_HSICALR+8.
But you can't just write something there: the register is write-protected. To remove the protection, use the CLK_HSIUNLCKR register.

In general, the procedure for changing the calibration value looks like this:

```
Записать 0xAC в CLK_HSIUNLCKR
Записать 0x35 туда-же. Такой хитрый способ отключения защиты от записи.
Записать нужное значение в CLK_HSITRIMR.
```

All these actions must be performed one after another, without being distracted by other operations — otherwise the trick will fail. After recording, CLK_HSITRIMR is blocked again.

Well? **Let's make up another useless example?**

We will use the same set: **indicator, LED, button** . But we will add a quartz to it — we need to arrange a clock failure to demonstrate CSS :)

We will now work not at 2 MHz, but at 16, so the value of our delay will have to be recalculated for the new realities:

```
void SomeDelay()
{
  for (unsigned long delay_count=0; delay_count<300000; delay_count++);
};
```

In order for a clock signal to be output to pin C4, it must be configured to output:

```
PC_DDR_bit.DDR4 = 1;
PC_CR1_bit.C14 = 1;
```

We'll disable the SYSCLK signal divider to work at maximum frequency.

```
CLK_CKDIVR = 0;
```

Remember that it is after all the generators, which means that if you don't turn it off, the clock signal from the quartz will also go through the divider. Do we need it?

After turning off the divider, blink the LED to show the delay duration at a frequency of 16 MHz:

```
for (char i=0; i<6; i++)
  {
   PD_ODR_bit.ODR4 ^= 1;
   SomeDelay();
  };
```

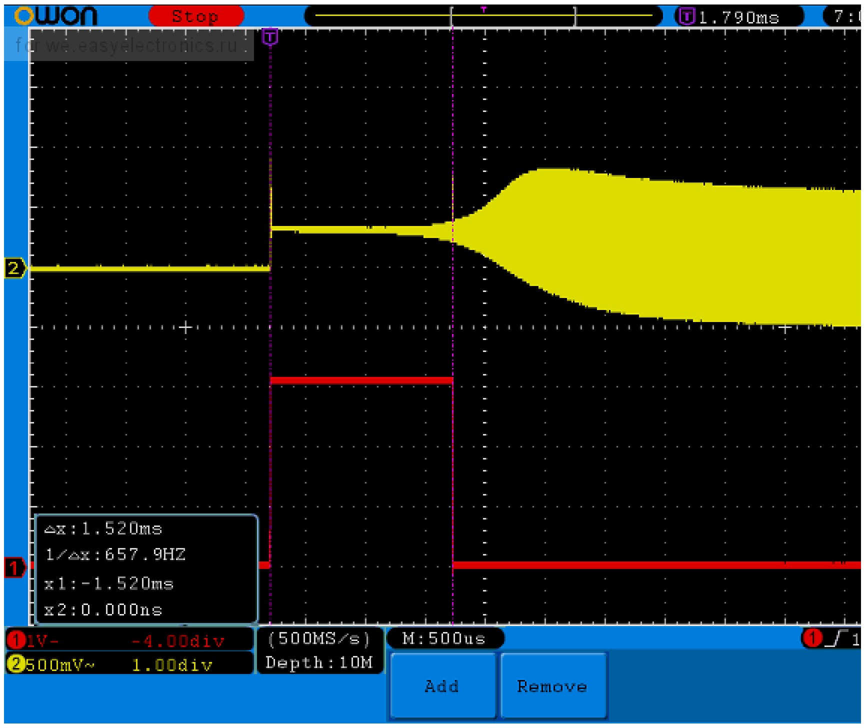
And then we wait for the button to be pressed and after that we start switching to HSE. In automatic mode

```
CLK_SWCR_bit.SWEN = 1;
CLK_SWR = 0x04;
```

After switching is started, we wait in a loop until the SWBSY flag drops

```
while (CLK_SWCR_bit.SWBSY==1);
```

Actually, it is not right to do this, at least because if HSE does not start, then we will hang here forever. It would be good to count the waiting time and start worrying if it is too long. By the way, how long does the HSE swing last at startup? In general, it depends on many factors, but for me with a 4 MHz quartz crystal, 18 pF load capacitors, and a supply voltage of 3 V, it turned out like this:



Yellow is a signal from one of the quartz pins (relative to the ground), and a red pulse with a duration just in our waiting cycle. **It turned out to be about 1.52 ms** .
Immediately after the switching is finished, we turn on the panic CSS and enable the interrupt from it:

```
CLK_CSSR_bit.CSSEN = 1;
CLK_CSSR_bit.CSSDIE = 1;
```

And in the CSS interrupt handler we will display the letter E on the indicator, like, ERROR.

```
ISR(CLK_interrupt, CLK_CSS_vector)
{
  PB_ODR = ~((1<<segment_G)|(1<<segment_F)|(1<<segment_E)|(1<<segment_D)
  CLK_CSSR_bit.CSSDIE = 0;
};
```

Then (we're back in main(), follow along) we blink the LED again (using the same delay function) so that we can see how the frequency has changed after switching to HSE.

And we turn on CCO, setting it to output a signal from HSE:

```
CLK_CCOR_bit.CCOSEL = 0x04;
```

Now you can arm yourself with a frequency meter and poke at pin **C4** , observing there a meander with the frequency of our quartz.

After everything is set up, in the cycle we catch the button press and change the divider to CCO:

```
while (1)
  {
   CLK_CCOR_bit.CCODIV = value;
   PB_ODR = numbers[value];

   while (PD_IDR_bit.IDR7 == 1);
   SomeShortDelay();
```

```
    value++;
    value &= 7;
  };
};
```

Sources

I haven't covered some of the clocking tricks. They are too specific and we'll cover them some other time. But if you're really interested, RM0031, page 87 is waiting for you!

Now that we know how to work with the clocking system and how to use interrupts, we can study, for example, timers. In the next part, we'll finally get rid of that idiotic cycle delay!

← Part 3 — Interrupts Contents Part 5 — Timers, the beginning →

stm8 , stm8l, css, cco, clocking

+13 21 February 2013, 17:03 **dcoder** 1

Files in topic: 4_CLK.zip

Comments (13)

RSS Collapse / Expand

Thank you, as always interesting! 0

sponge_bob
21 February 2013, 18:32

Thank you 0

dcoder
21 February 2013, 18:50 ↑

> *We cannot switch back to quartz (maybe the failure was temporary, and in a couple of seconds the quartz will start normally), which means that although the device continues to work, it will not be able to return to full functionality until it reboots.*

We can reset the MK programmatically and reconfigure the generator.
When a failure occurs, the severity is unknown. Maybe the failure occurred during the exchange via UART or another interface, more critical to timings, and therefore the data turned out to be broken. Initialization from scratch is more reliable in this sense, since it is assumed that at the beginning our state is undefined.
PS: And this, when the protection is triggered, the divider is reset to 8, or am I mistaken? 0

angel5a
21 February 2013, 18:45

No, it doesn't seem to reset. rm0031 doesn't say anything.

It would be better to give the option to start the generator and see if it works or not without switching to it. And then (for example, if it hasn't failed in 10 seconds) - you can switch back. Software reset is a crutch and not always convenient. 0

dcoder
21 February 2013, 18:50 ↑

Why is the CSSDGON bit not illuminated? 0

Vga
21 February 2013, 20:04

Thank you) It was very interesting to read about CSS. 0

Will there be something about DMA soon?

KnifeMaster
21 February 2013, 20:36

About DMA will not be very soon :) I have not yet decided exactly where to put it, but it will definitely be after the ADC. And the ADC after the timers. And there are at least two articles about them.

+1

dcoder
21 February 2013, 21:25 ↑

> *I haven't told you about some of the timing tricks. They are too specific and we'll look at them some other time. But if you're really interested, RM0031, page 87 is waiting for you!*

Better specify the number and name of the section. There are different versions of Refmans. Mine starts with section 9. Clock. Or are you suggesting to read it all?)

0

Vga
22 February 2013, 00:29

Exactly the whole thing :) Because, alas, there is no section "what d didn't write about" :(

0

dcoder
22 February 2013, 00:53 ↑

There are just some things in this section that are more related to all sorts of peripherals (RTC, LCD, unnecessary BEEP). So I'll tell you about them when the opportunity arises)

0

dcoder
22 February 2013, 00:54 ↑

> *PC_DDR_bit.DDR4 = 1;*
> *PC_CR1_bit.C14 = 1;*

Is there an error here? Bit C14? Or C4?

0

Nastik-kum
02 October 2013, 13:43

Okay, I found it =)

0

Nastik-kum
02 October 2013, 14:59 ↑

This often leaves me stumped too. They could have done something like C1_4

0

dcoder
05 October 2013, 01:53 ↑

Only registered and authorized users can leave comments.