STM8S + SDCC: Programming WITHOUT SPL. Interfaces: UART in transmitter mode, ADC in single-sample mode, I2C in master mode

(/2018/stm8_uart_adc_i2c/)

sections: STM8 (/tags/stm8) , UART (/tags/uart) , I2C , date: May 5 , (/tags/i2c) 2018

This time we will talk about the hardware interfaces of STM8S: UART, ADC and I ^{2} C. Each of these interfaces supports several operating modes, but now I would like to focus on the most typical, in my opinion, examples of their use: a) organizing a transmitter on UART, b) single ADC measurement mode, c) using I ^{2} C in master mode. Let me remind you that I gave the example of using SPI in master mode using the driver for a 4-digit seven-segment indicator (/2018/stm8_assembler/#8) .

The documentation you will need to read this article: Reference Manual STM8S - RM0016 (http://www.st.com/content/ccc/resource/technical/document/reference_manual/9a/1b/85/07/ca/eb/4f/dd/CD00190271.pdf/files/CD00190271.pdf/jcr: , chapters: 22 (UART), 24 (ADC), 21 (I ^{2} C). I will use the 20-pin STM8S103F3P6 as the target microcontroller.

Just like last time, the emphasis will be on "pure" programming in C and Assembler without using third-party libraries. The open-source SDCC version 3.7 is used as a compiler. In fairness, I will note that I introduced macros for direct access to bit instructions in order to at least somehow optimize the code.

You can download the full source code with assembly files and compiled firmware using the link at the end of the article.

The article uses formulas in MathML format, which is supported by the Firefox browser. For Chrome and Opera browsers, you will need to install the MathML extension of the same name.

Content:

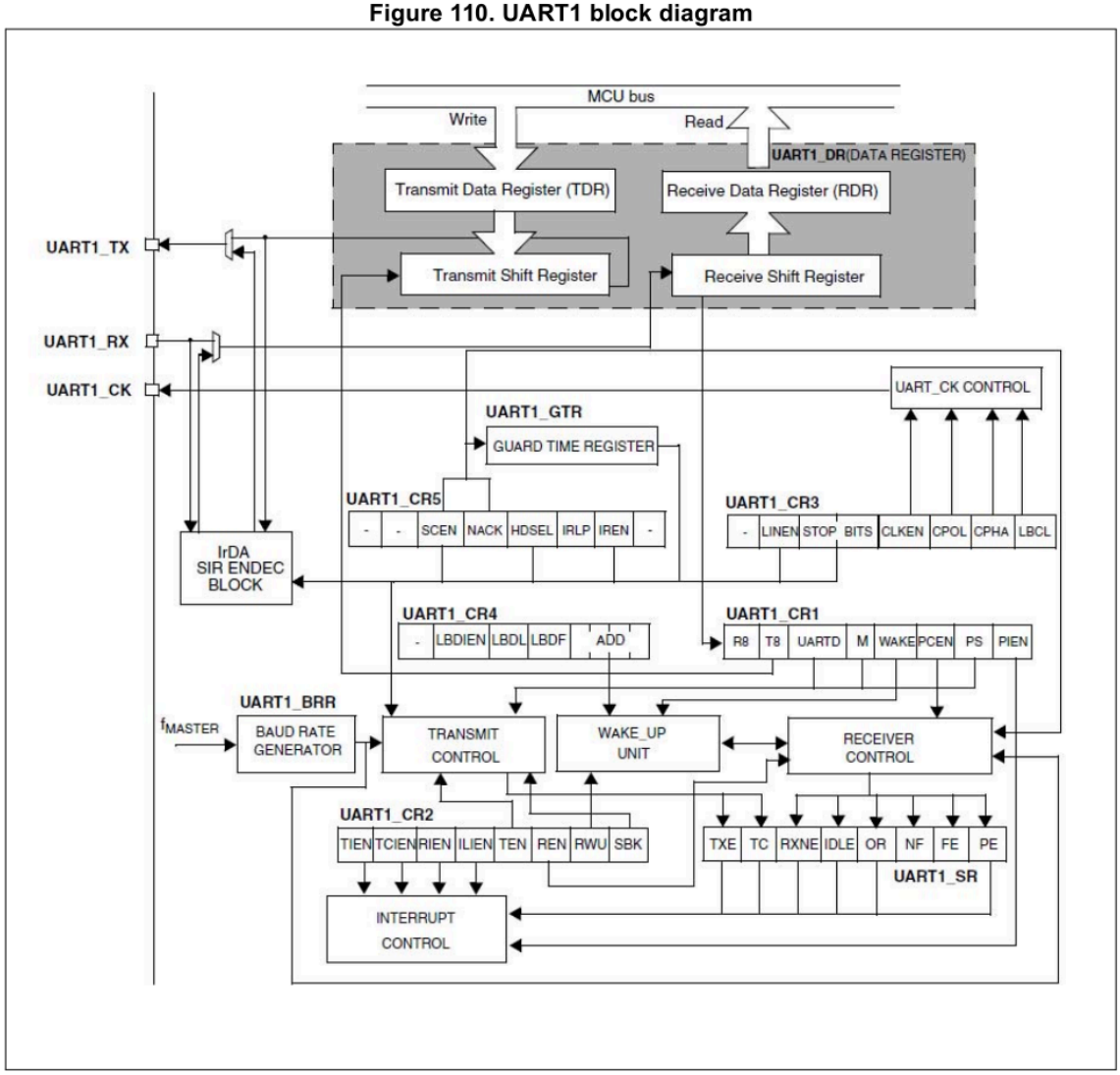
Note from 09/01/2022: In SDCC version 4.2, the format of passing function arguments has changed. If previously all arguments were passed via the stack, now they are passed via registers. Therefore, for compatibility with old code, the compilation option "--sdcccall 0" should be added.

1. UART1 transmitter with baud rate 921600

The documentation for the STM8S103f3 chip promises us a UART protocol speed of ~1 Mbit/sec at a frequency of f $_{MASTER}$ =16 MHz. I would like to note that the declared speed is also available without an external quartz crystal or generator.

The block diagram of the UART1 module is shown in the picture below:

Figure 110. UART1 block diagram



(/img/stm8/uart/uart1_01.png)

The first thing that catches your eye here is the data register - UART1_DR. As you can see, it consists of two independent registers for reading and writing, and also two independent internal shift registers. That is, when writing to UART1_DR and reading from it, you are actually accessing different registers.

UART1 includes one flag register UART1_SR1 and five configuration registers: UART1_CR1, UART1_CR2, UART1_CR3, UART1_CR4, UART1_CR5. In addition to the data register UART1_DR, there are also a pair of registers for specifying the bit rate: UART1_BRR1 and UART1_BRR2, a prescaler register: UART1_PSCR, and a protective register UART1_GTR for Smartcard mode.

The general map of registers with preset values is shown in the following table:

Table 61. UART1 register map

Address	Register name	7	6	5	4	3	2	1	0
0x00	UART1_SR Reset Value	TXE 1	TC 1	RXNE 0	IDLE 0	OR 0	NF 0	FE 0	PE 0
0x01	UART1_DR Reset Value	DR7 x	DR6 x	DR5 x	DR4 x	DR3 x	DR2 x	DR1 x	DR0 x
0x02	UART1_BRR1 Reset Value	UART_DIV[11:4] 00000000							
0x03	UART1_BRR2 Reset Value	UART_DIV[15:12] 0000				UART_DIV[3:0] 0000			
0x04	UART1_CR1 Reset Value	R8 0	T8 0	UARTD 0	M 0	WAKE 0	PCEN 0	PS 0	PIEN 0
0x05	UART1_CR2 Reset Value	TIEN 0	TCIEN 0	RIEN 0	ILIEN 0	TEN 0	REN 0	RWU 0	SBK 0
0x06	UART1_CR3 Reset Value	- 0	LINEN 0	STOP 00		CKEN 0	CPOL 0	CPHA 0	LBCL 0
0x07	UART1_CR4 Reset Value	- 0	LBDIEN 0	LBDL 0	LBDF 0	ADD[3:0] 0000			
0x08	UART1_CR5 Reset Value	- 0	- 0	SCEN 0	NACK 0	HDSEL 0	IRLP 0	IREN 0	0
0x09	UART1_GTR Reset Value	GT7 0	GT6 0	GT5 0	GT4 0	GT3 0	GT2 0	GT1 0	GT0 0
0x0A	UART1_PSCR Reset Value	PSC7 0	PSC6 0	PSC5 0	PSC4 0	PSC3 0	PSC2 0	PSC1 0	PSC0 0

(/img/stm8/uart/uart1_02.png)

I discussed the register values in the article STM8S+SDCC+SPL: Using the UART1 Module Using the Printf() Function as an Example (/2016/stm8_spl_printf/) . In order to configure and run the UART1 transmitter, you only need to keep in mind three flags.

In UART1_CR1 this is the UARTD flag that turns the UART1 module on and off. Note that when the flag is reset, the UART1 module *is on* :

22.7.5 Control register 1 (UART_CR1)

Address offset: 0x04

Reset value: 0x00

7	6	5	4	3	2	1	0
R8	T8	UARTD	M	WAKE	PCEN	PS	PIEN
rw	rw	rw	rw	rw	rw	rw	rw

Bit 5 **UARTD**: *UART Disable (for low power consumption).*
When this bit is set the UART prescaler and outputs are stopped at the end of the current byte transfer in order to reduce power consumption. This bit is set and cleared by software.
0: UART enabled
1: UART prescaler and outputs disabled

(/img/stm8/uart/uart1_04.png)

In the UART1_CR2 register we will be interested in the TEN flag which enables the transmitter:

22.7.6 Control register 2 (UART_CR2)

Address offset: 0x05

Reset value: 0x00

7	6	5	4	3	2	1	0
TIEN	TCIEN	RIEN	ILIEN	TEN	REN	RWU	SBK
rw	rw	rw	rw	rw	rw	rw	rw

Bit 3 **TEN**: Transmitter enable (1) (2)
This bit enables the transmitter. It is set and cleared by software.
0: Transmitter is disabled
1: Transmitter is enabled

(/img/stm8/uart/uart1_05.png)

And in the flag register UART1_SR1 we will be interested in the TXE bit, which is set at the moment of transferring the contents of UART1_DR to the shift register. That is, at the moment when UART1_DR is released. This flag is cleared by writing to UART1_DR.

22.7.1 Status register (UART_SR)

Address offset: 0x00

Reset value: 0xC0

7	6	5	4	3	2	1	0
TXE	TC	RXNE	IDLE	OR/LHE	NF	FE	PE
r	rc_w0	rc_w0	r	r	r	r	r

Bit 7 **TXE**: Transmit data register empty

This bit is set by hardware when the content of the TDR register has been transferred into the shift register. An interrupt is generated if the TIEN bit =1 in the UART_CR2 register. It is cleared by a write to the UART_DR register.

0: Data is not transferred to the shift register

1: Data is transferred to the shift register

(/img/stm8/uart/uart1_03.png)

В качестве шаблона проекта я взял пример из предыдущей статьи: Передача параметров из Си в ассемблерную функцию (/2018/stm8_assembler/#4)

As a result, I got the following source code:

```
#include <stdint.h>
#include "stm8s103f.h"
#include "uart1.h"

#define LED 5

extern void delay(uint16_t value);

int main( void ) {
    uint16_t i=0;
    //----------- Setup Clock ----------------------
    // fCPU =16MHz
    CLK_CKDIVR=0;
    // enable UART and turn off other  peripherals
    CLK_PCKENR1=0;
    CLK_PCKENR2= 0 ;
    clk_pckenr1_bset(#3);   // enable UART1
    //----------- Setup GPIO -----------------------
    // GPIO setup
    pb_ddr_bset(#LED);
    pb_cr1_bset(#LED);
    //----------- Setup UART1 ----------------------
    // Clear
    UART1_CR1=0;
    UART1_CR2= 0 ;
    UART1_CR3=0;
    UART1_CR4=0;
    UART1_CR5=0;
    UART1_GTR=0;
    UART1_PSCR=0;
    // setup UART1
    uart1_cr1_bset(#5);     // set UARTD, UART1 disable
    // 9600 Baud Rate, when fMASTER=16MHz
    //UART1_BRR2=0x03
    //UART1_BRR1=0x68
    // 115200 Baud Rate, when fMASTER=16MHz
    //UART1_BRR2=0x0b
    //UART1_BRR1=0x08
    // 230400 Baud Rate, when fMASTER=16MHz
    //UART1_BRR2=0x05;
    //UART1_BRR1=0x04;
    // 460800 Baud Rate, when fMASTER=16MHz
    //UART1_BRR2=0x03;
    //UART1_BRR1=0x02;
    // 921600 Baud Rate, when fMASTER=16MHz
    UART1_BRR2=0x01;
    UART1_BRR1=0x01;
    // Trasmission Enable
    uart1_cr2_bset(#3);      // set TEN, Transmission Enable
    // enable UART1
    uart1_cr1_bres(#5);      // clear UARTD, UART1 enable
    // main loop
    for(;;)
    {
        static uint16_t i=0;
        pb_odr_bcpl(#LED);
        uart1_print_string("count: ");
        uart1_print_number(i++);
        uart1_send_char('\n');
        delay(1000);
    }
}
```

Macros: register_bset(), register_bres(), register_bcpl() are defined in the header file stm8s103f.h. They are replaced by assembler instructions:

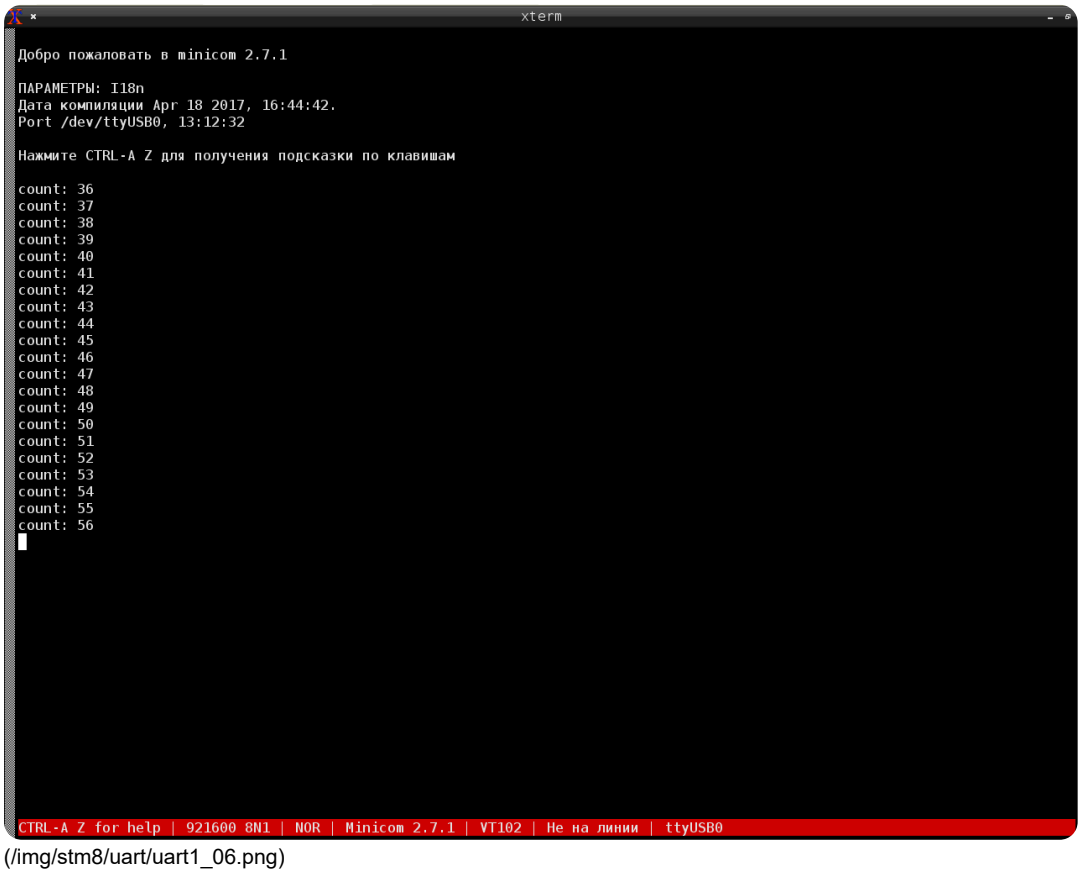
```
;   main.c: 51: uart1_cr2_bset(#3);      // set TEN, Transmission Enable
    bset    0x5235,#3
;   main.c: 53: uart1_cr1_bres(#5);      // clear UARTD, UART1 enable
    bres    0x5234,#5
00102$:
;   main.c: 58: pb_odr_bcpl(#LED);
    bcpl    0x5005,#5
```

I didn't use the printf() function this time, it's too heavy. So I had to replace it with several self-written functions for transmitting data via UART:

```
#include "uart1.h"

#define uart1_sr1_btjf(X,Y) __asm Y: btjf 0x5230,X,Y __endasm
#define len 6

void uart1_send_char(uint8_t ch) {
    UART1_DR=ch;
    uart1_sr1_btjf(#7,l0);      // while (TXE bit == 0);
}

void uart1_print_string(char *str) {
    while (*str)
    {
        uart1_send_char(*str++);
    }
}

void uart1_print_number(uint16_t num){
    uint8_t n[len];
    char *s=n+(len-1);
    *s=0;           // EOL
    do {
        *(--s)=(num% 10 + 0x30 );
        num=num/ 10 ;
    } while (num>0);

    uart1_print_string(s);
}

void uart1_print_bcd(uint8_t num) {
    UART1_DR=(num>>4) + 0x30;
    uart1_sr1_btjf(#7,l1);      // while (TXE bit == 0);
    UART1_DR=(num & 0x0f) + 0x30;
    uart1_sr1_btjf(#7,l2);      // while (TXE bit == 0);
}
```

The firmware weighs 284 bytes. You can view the output in any terminal program, for example in Linux using minicom:

```
$ minicom -D /dev/ttyUSB0 -b 921600
```



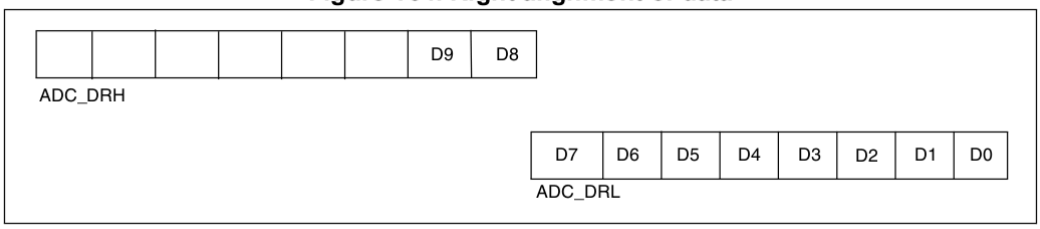
(/img/stm8/uart/uart1_06.png)

or via the Arduino IDE by calling the serial monitor.

2. ADC in single-sample mode

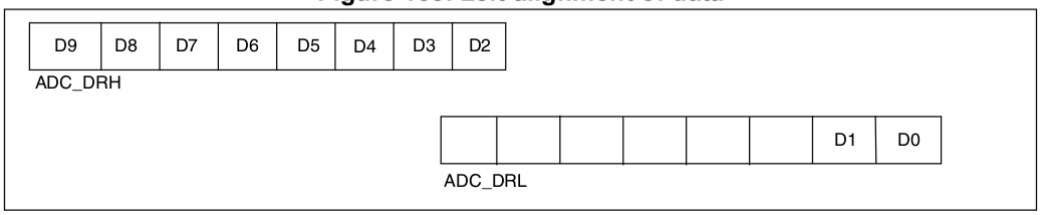
The STM8S has a 10-bit ADC, and since the microcontroller itself is 8-bit, there are options for aligning the result to get a single-byte 8-bit number or a two-byte 10-bit number. The alignment options are shown in the picture below:

Figure 164. Right alignment of data



Left Alignment: 8 Most Significant bits are written in the ADC_DH register, then the remaining Least Significant bits are written in the ADC_DL register. The Most Significant Byte must be read first followed by the Least Significant Byte.

Figure 165. Left alignment of data



(/img/stm8/uart/uart1_07.png)

It is easy to guess that left-alignment is convenient for obtaining an 8-bit ADC value, and right-alignment is convenient for obtaining a 10-bit value.

If we talk about the single measurement mode, then in addition to the above-mentioned register pair ADC_DR, we will be interested in several more RVV:

1. Flag - configuration register ADC_CSR:

24.11.3 ADC control/status register (ADC_CSR)

Address offset: 0x20

Reset value: 0x00

7	6	5	4	3	2	1	0
EOC	AWD	EOCIE	AWDIE	CH[3:0]			
rw	rc_w0	rw	rw	rw	rw	rw	rw

Bit 7 **EOC**: End of conversion
This bit is set by hardware at the end of conversion. It is cleared by software by writing ‘0’.
0: Conversion is not complete
1: Conversion complete

Bit 6 **AWD**: Analog Watchdog flag
0: No analog watchdog event
1: An analog watchdog event occurred. In buffered continuous or scan mode you can read the ADC_AWSR register to determine the data buffer register related to the event. An interrupt request is generated if AWDIE=1.
Note: This bit is not available for ADC2

Bit 5 **EOCIE**: Interrupt enable for EOC
This bit is set and cleared by software. It enables the interrupt for End of Conversion.
0: EOC interrupt disabled
1: EOC interrupt enabled. An interrupt is generated when the EOC bit is set.

Bit 4 **AWDIE**: Analog watchdog interrupt enable
0: AWD interrupt disabled.
1: AWD interrupt enabled
Note: This bit is not available for ADC2

Bits 3:0 **CH[3:0]**: Channel selection bits
These bits are set and cleared by software. They select the input channel to be converted.
0000: Channel AIN0
0001: Channel AIN1
....
1111: Channel AIN15

(/img/stm8/uart/uart1_08.png)

The fields of interest here are EOC - conversion complete, EOCIE - enable interrupt on conversion complete, and CH[3:0] - analog channel select.

2. The first ADC configuration register - ADC_CR1:

24.11.4 ADC configuration register 1 (ADC_CR1)

Address offset: 0x21

Reset value: 0x00

7	6	5	4	3	2	1	0
Reserved	SPSEL[2:0]			Reserved		CONT	ADON
r	rw	rw	rw	r		rw	rw

Bit 7 Reserved, always read as 0.

Bits 6:4 **SPSEL[2:0]**: Prescaler selection

These control bits are written by software to select the prescaler division factor.

000: $f_{ADC} = f_{MASTER}/2$
001: $f_{ADC} = f_{MASTER}/3$
010: $f_{ADC} = f_{MASTER}/4$
011: $f_{ADC} = f_{MASTER}/6$
100: $f_{ADC} = f_{MASTER}/8$
101: $f_{ADC} = f_{MASTER}/10$
110: $f_{ADC} = f_{MASTER}/12$
111: $f_{ADC} = f_{MASTER}/18$

See *Section 24.5.2 on page 426*.

Note: It is recommended to change the SPSEL bits when ADC is in power down. This is because internally there can be a glitch in the clock during this change. Otherwise the user is required to ignore the 1st converted result if the change is done when ADC is not in power down.

Bits 3:2 Reserved, always read as 0.

Bit 1 **CONT**: Continuous conversion

This bit is set and cleared by software. If set, conversion takes place continuously till this bit is reset by software.

0: Single conversion mode
1: Continuous conversion mode

Bit 0 **ADON**: A/D Converter on/off

This bit is set and reset by software. This bit must be written to wake up the ADC from power down mode and to trigger the start of conversion. If this bit holds a value of 0 and a 1 is written to it then it wakes the ADC from power down mode. Conversion starts when this bit holds a value of 1 and a 1 is written to it. As soon as the ADC is powered on, the output stage of the selected channel is disabled.

0: Disable ADC conversion/calibration and go to power down mode.
1: Enable ADC and to start conversion

(/img/stm8/uart/uart1_09.png)

Here, the ADC sampling frequency is set via SPSEL[2:0]. The CONT bit sets or resets the continuous operation mode of the ADC. ADON enables or disables the ADC.

3. The second ADC configuration register - ADC_CR2:

24.11.5 ADC configuration register 2 (ADC_CR2)

Address offset: 0x22

Reset value: 0x00

7	6	5	4	3	2	1	0
Reserved	EXTTRIG	EXTSEL[1:0]		ALIGN	Reserved	SCAN	Reserved
r	rw	rw	rw	rw	r	rw	r

Bit 7 Reserved, must be kept cleared.

Bit 6 **EXTTRIG**: External trigger enable

This bit is set and cleared by software. It is used to enable an external trigger to trigger a conversion.

0: Conversion on external event disabled

1: Conversion on external event enabled

Note: To avoid a spurious trigger event, use the BSET instruction to set EXTTRIG without changing other bits in the register.

Bits 5:4 **EXTSEL[1:0]**: External event selection

The two bits are written by software. They select one of four types of event used to trigger the start of ADC conversion.

00: Internal TIM1 TRGO event

01: External interrupt on ADC_ETR pin

10: Reserved

11: Reserved

Bit 3 **ALIGN**: Data alignment

This bit is set and cleared by software.

0: Left alignment (the eight MSB bits are written in the ADC_DRH register then the remaining LSB bits are written in the ADC_DRL register). The reading order should be MSB first and then LSB.

1: Right alignment (eight LSB bits are written in the ADC_DRL register then the remaining MSB bits are written in the ADC_DH register). The reading order should be LSB first and then MSB.

Note: The ALIGN bit influences the ADC_DRH/ADC_DRL register reading order and not the reading order of the buffer registers.

Bit 2 Reserved, must be kept cleared.

Bit 1 **SCAN**: Scan mode enable

This bit is set and cleared by software.

0: Scan mode disabled

1: Scan mode enabled

(/img/stm8/uart/uart1_10.png)

Here we will be interested only in ALIGN - the alignment flag.

Also, don't forget about the registers that disable Schmitt triggers (https://ru.wikipedia.org/wiki/Триггер_Шмитта) :

24.11.9 ADC Schmitt trigger disable register high (ADC_TDRH)

Address offset: 0x26

Reset value: 0x00

7	6	5	4	3	2	1	0
TD[15:8]							
rw	rw	rw	rw	rw	rw	rw	rw

Bits 7:0 **TD[15:8]** Schmitt trigger disable high

These bits are set and cleared by software. When a TDx bit is set, it disables the I/O port input Schmitt trigger of the corresponding ADC input channel x even if this channel is not being converted. This is needed to lower the static power consumption of the I/O port.
0: Schmitt trigger enabled
1: Schmitt trigger disabled

24.11.10 ADC Schmitt trigger disable register low (ADC_TDRL)

Address offset: 0x27

Reset value: 0x00

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

Bits 7:0 **TD[7:0]** Schmitt trigger disable low

These bits are set and cleared by software. When a TDx bit is set, it disables the I/O port input Schmitt trigger of the corresponding ADC input channel x even if this channel is not being converted. This is needed to lower the static power consumption of the I/O port.
0: Schmitt trigger enabled
1: Schmitt trigger disabled

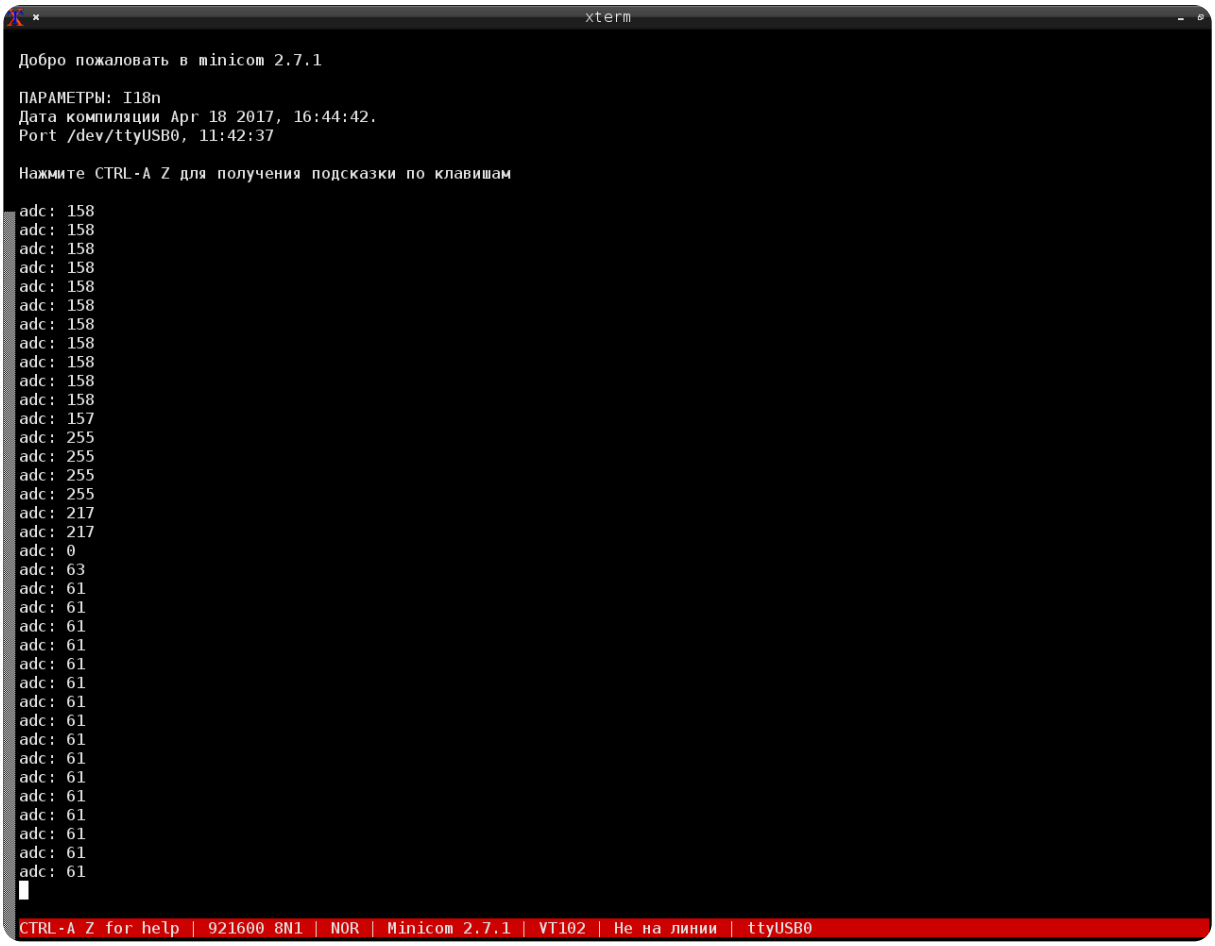
(/img/stm8/uart/uart1_11.png)

Here, each bit corresponds to its own channel. Disabling the trigger makes it impossible to use the corresponding pin as a digital input/output pin, and leaving it on significantly reduces the accuracy of the ADC measurement. The trigger is disabled by writing one.

The program version for receiving the 8-value from the ADC and outputting it via UART turned out to be like this:

```
#include <stdint.h>
#include "stm8s103f.h"
#include "uart1.h"

#define LED 5

extern void delay(uint16_t value);
volatile uint8_t adc_value;

void adc_irq(void) __interrupt(22) {
    adc_value=ADC_DRH;
    adc_csr_bres(#7);       // Clear EOC bit
}

void awu_irq(void) __interrupt(1) {
    awu_csr_bres(# 5 );        //Clear AWUF bit for AWU_CSR
}

int main( void ) {
    //----------- Setup Clock ----------------------
    // fCPU =16MHz
    CLK_CKDIVR=0;
    // enable UART and turn off other  peripherals
    CLK_PCKENR1=0;
    CLK_PCKENR2= 0 ;
    clk_pckenr1_bset(#3);   // enable UART1
    clk_pckenr2_bset(#2);   // enable AWU
    clk_pckenr2_bset(#3);   // enable ADC
    //----------- Setup GPIO -----------------------
    // GPIO setup
    pb_ddr_bset(#LED);
    pb_cr1_bset(#LED);
    // ------- Setup AWU & Buzzer ------------------------
    AWU_TBR = 12;
    AWU_APR = 62 ;        // = 1 sec
    AWU_CSR = 0x10 ;      // set AWUEN bit for AWUCSR

    BEEP_CSR=0x1e;      // buuuuuuuuuzzzzz (low tone)
    beep_csr_bset(#5);  // Enable BEEP
    delay(100);
    beep_csr_bres(#5);  // Disable BEEP
    //----------- Setup UART1 ----------------------
    // Clear
    UART1_CR1=0;
    UART1_CR2= 0 ;
    UART1_CR3=0;
    UART1_CR4=0;
    UART1_CR5=0;
    UART1_GTR=0;
    UART1_PSCR=0;
    // setup UART1
    uart1_cr1_bset(#5);     // set UARTD, UART1 disable
    // 9600 Baud Rate, when fMASTER=16MHz
    //UART1_BRR2=0x03
    //UART1_BRR1=0x68
    // 115200 Baud Rate, when fMASTER=16MHz
    //UART1_BRR2=0x0b
    //UART1_BRR1=0x08
    // 230400 Baud Rate, when fMASTER=16MHz
    //UART1_BRR2=0x05;
    //UART1_BRR1=0x04;
    // 460800 Baud Rate, when fMASTER=16MHz
    //UART1_BRR2=0x03;
    //UART1_BRR1=0x02;
    // 921600 Baud Rate, when fMASTER=16MHz
    UART1_BRR2=0x01;
    UART1_BRR1=0x01;
    // Trasmission Enable
    uart1_cr2_bset(#3);      // set TEN, Transmission Enable
    // enable UART1
    uart1_cr1_bres(#5);      // clear UARTD, UART1 enable
    //  ------------- ADC Setup --------------------
    adc_cr2_bres(#3);       // Clear ALIGN, Left  Align
    adc_cr1_bres(#1);       // Clear CONT, Single Conersion Mode
    ADC_CSR &= 0xf0;        // Clear CH[3:0] bits
    adc_csr_bset(#1);       // Select AIN2 Channel
    ADC_CR1 |= 0x70;        // Prescaler = fMASTER/18
    adc_cr2_bres(#6);       // Clear EXTTRIG, Disable External Trigger
    ADC_TDRL=0x04;          // Diable Schmitt Trigger
    adc_csr_bset(#5);       // Set EOCIE, enable ADC Interrupt
    // ------------ END SETUP ----------------------
    // main loop
    rim();                  // Enable Interrupts
    for(;;)
    {
        adc_cr1_bset(#0);   // set ADON, Start Of Conversion
        wfi();              // sleep
        pb_odr_bcpl(#LED);
        uart1_print_string("adc: ");
        uart1_print_number((uint16_t)adc_value);
        uart1_send_char('\n');
        delay(1000);
    }
}
```

I connected a potentiometer to the PC3 pin and after turning the knob a couple of times I got the following result:



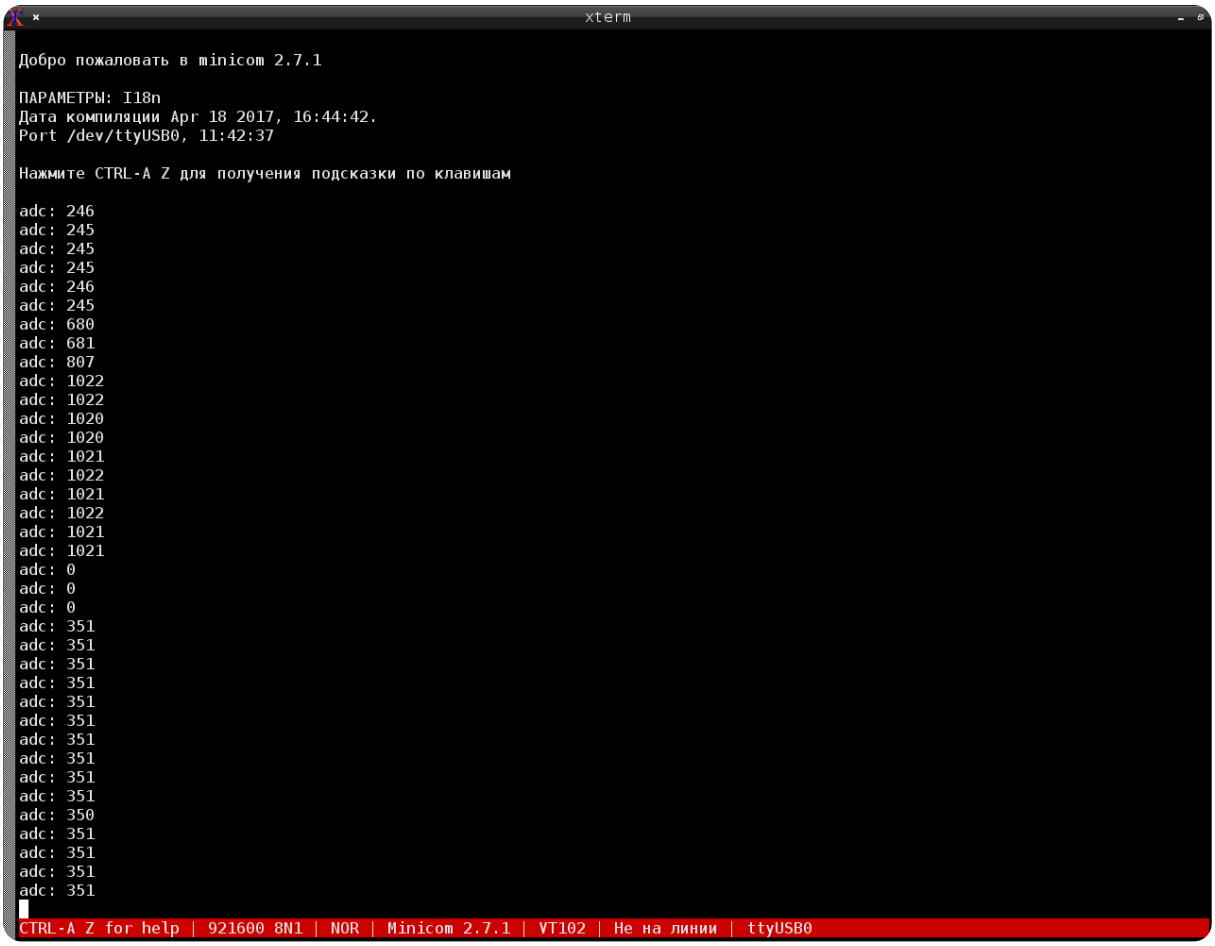
(/img/stm8/uart/uart1_12.png)

As you can see, it works quite well, does not make noise, but there is a "pitfall" in the program. If you comment out the line: AWU_CSR = 0x10, rebuild and re-upload the firmware, it will not work. The microcontroller will not exit sleep mode after the wfi instruction. Trying to replace the wfi instruction with a cycle of waiting for the EOC flag to be set will not help. At the same time, if you comment out the wfi instruction this time, the program seems to work. What is the relationship between AWU and the ADC, I honestly do not know. Perhaps the analog watchdog is affected somehow. I can only say with confidence that the problem is not in the microcontroller. A similar program in assembler on the same microcontroller works without any AWU, but when transferring the algorithm to C, such a problem arose. So I assume that the problem is in the SDCC compiler, which is very sad.

The program version for 10-bit ADC is presented below:

```
#include <stdint.h>
#include "stm8s103f.h"
#include "uart1.h"

#define LED 5

extern void delay(uint16_t value);
volatile uint16_t adc_value;

void adc_irq(void) __interrupt(22) {
    adc_value=ADC_DRL;
    adc_value|=(ADC_DRH<<8);
    adc_value &= 0x03ff;
    adc_csr_bres(#7);       // Clear EOC bit
}

void awu_irq(void) __interrupt(1) {
    awu_csr_bres(# 5 );        //Clear AWUF bit for AWU_CSR
}

int main( void ) {
    //----------- Setup Clock ----------------------
    // fCPU =16MHz
    CLK_CKDIVR=0;
    // enable UART and turn off other  peripherals
    CLK_PCKENR1=0;
    CLK_PCKENR2= 0 ;
    clk_pckenr1_bset(#3);   // enable UART1
    clk_pckenr2_bset(#2);   // enable AWU
    clk_pckenr2_bset(#3);   // enable ADC
    //----------- Setup GPIO -----------------------
    // GPIO setup
    pb_ddr_bset(#LED);
    pb_cr1_bset(#LED);
    // ------- Setup AWU & Buzzer ------------------------
    AWU_TBR = 12;
    AWU_APR = 62 ;        // = 1 sec
    AWU_CSR = 0x10 ;      // set AWUEN bit for AWUCSR

    BEEP_CSR=0x1e;      // buuuuuuuuuzzzzz (low tone)
    beep_csr_bset(#5);  // Enable BEEP
    delay(100);
    beep_csr_bres(#5);  // Disable BEEP
    //----------- Setup UART1 ----------------------
    // Clear
    UART1_CR1=0;
    UART1_CR2= 0 ;
    UART1_CR3=0;
    UART1_CR4=0;
    UART1_CR5=0;
    UART1_GTR=0;
    UART1_PSCR=0;
    // setup UART1
    uart1_cr1_bset(#5);     // set UARTD, UART1 disable
    // 9600 Baud Rate, when fMASTER=16MHz
    //UART1_BRR2=0x03
    //UART1_BRR1=0x68
    // 115200 Baud Rate, when fMASTER=16MHz
    //UART1_BRR2=0x0b
    //UART1_BRR1=0x08
    // 230400 Baud Rate, when fMASTER=16MHz
    //UART1_BRR2=0x05;
    //UART1_BRR1=0x04;
    // 460800 Baud Rate, when fMASTER=16MHz
    //UART1_BRR2=0x03;
    //UART1_BRR1=0x02;
    // 921600 Baud Rate, when fMASTER=16MHz
    UART1_BRR2=0x01;
    UART1_BRR1=0x01;
    // Trasmission Enable
    uart1_cr2_bset(#3);      // set TEN, Transmission Enable
    // enable UART1
    uart1_cr1_bres(#5);      // clear UARTD, UART1 enable
    //  ------------- ADC Setup --------------------
    adc_cr2_bset(#3);       // Set ALIGN, Right  Align
    adc_cr1_bres(#1);       // Clear CONT, Single Conersion Mode
    ADC_CSR &= 0xf0;        // Clear CH[3:0] bits
    adc_csr_bset(#1);       // Select AIN2 Channel
    ADC_CR1 |= 0x70;        // Prescaler = fMASTER/18
    adc_cr2_bres(#6);       // Clear EXTTRIG, Disable External Trigger
    ADC_TDRL=0x04;          // Diable Schmitt Trigger
    adc_csr_bset(#5);       // Set EOCIE, enable ADC Interrupt
    // ------------ END SETUP ----------------------
    // main loop
    rim();                  // Enable Interrupts
    for(;;)
    {
        adc_cr1_bset(#0);   // set ADON, Start Of Conversion
        wfi();              // sleep
        pb_odr_bcpl(#LED);
        uart1_print_string("adc: ");
        uart1_print_number((uint16_t)adc_value);
        uart1_send_char('\n');
        delay(1000);
    }
}
```

The result of working with the same potentiometer:



(/img/stm8/uart/uart1_13.png)

In my opinion, for a 10-bit ADC this is quite a decent result. Only the last bit is noisy, and even then, only slightly.

3. I ^{2}C in master mode, using RTC DS1307/DS3231 as an example. Initialization

Hardware I ^{2}C interface in STM8:

- ensures bus operation at speeds of 100 kHz (standard) and 400 kHz (fast);
- can work with both 7-bit I ^{2}C addresses and 10-bit ones.

In order to form a full opinion about the capabilities of the I ^{2}C module, you will need to familiarize yourself with its configuration registers. Personally, I liked the module, it is a powerful and flexible thing. Now I would like to consider the simplest mode of its operation: master mode at a standard speed of 100 kHz and a 7-bit I ^{2}C address, without using interrupts (operation in polling mode). As a slave, I chose RTC DS3231, which probably everyone is already familiar with.

I would divide the order of work with I ^{2}C into initialization and working cycle. I think that it makes sense to first consider the registers that are responsible for initialization.

In this case, initialization takes 8 lines of code and affects 5 registers:

```
    //----------- Setup I2C ------------------------
    i2c_cr1_bres(#0);       // PE=0, disable I2C before setup
    I2C_FREQR= 16;          // peripheral frequence =16MHz
    I2C_CCRH = 0;           // =0
    I2C_CCRL = 80;          // 100kHz for I2C
    i2c_ccrh_bres(#7);      // set standart mode(100кHz)
    i2c_oarh_bres(#7);      // 7-bit address mode
    i2c_oarh_bset(#6);      // see reference manual
    i2c_cr1_bset(#0);       // PE=1, enable I2C
    //------------- End Setup ---------------------
```

Initialization of the I ^{2}C module consists of setting the operating speed and switching to the desired mode: standard bus speed, 7-bit address. **Switching to the master mode and back to the slave occurs automatically** , after the START and STOP commands.

I2C module registers:

1. The I2C_FREQR register is used to specify the f $_{MASTER}$ frequency of the microcontroller. Do not forget that the microcontroller can be clocked from an external quartz or generator, the frequency of which the microcontroller itself cannot know. It should be specified here. In this case, the f $_{MASTER}$ frequency for the standard bus speed should be at least 1 MHz, and at least 4 MHz for the 400 kHz "fast"

21.7.3 Frequency register (I2C_FREQR)

Address offset: 0x02
Reset value: 0x00

7	6	5	4	3	2	1	0
Reserved		FREQ[5:0]					
r		rw					

Bits 7:6 Reserved

Bits 5:0 **FREQ[5:0]** Peripheral clock frequency. (1)

The FREQ field is used by the peripheral to generate data setup and hold times compliant with the I2C specifications. The FREQ bits must be programmed with the peripheral input clock frequency value:
The allowed range is between 1 MHz and 24 MHz
000000: not allowed
000001: 1 MHz
000010: 2 MHz
...
011000: 24 MHz
Higher values: not allowed

1. The minimum peripheral clock frequencies for respecting the I^2C bus timings are:
1 MHz for standard mode and 4 MHz for fast mode

mode.

(/img/stm8/uart/uart1_14.png)

2. The operating frequency of the I 2 C bus is set using the CCR number in the register pair I2C_CCRH:I2C_CCRL:

21.7.11 Clock control register low (I2C_CCRL)

Address offset: 0x02
Reset value: 0x0B

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

Bits 7:0 **CCR[7:0]** Clock control register (Master mode)
Controls the SCLH clock in Master mode.
– Standard mode:
Period(I2C) = 2 * CCR * t_{MASTER}
t_{high} = CCR * t_{MASTER}
t_{low} = CCR * t_{MASTER}
– Fast mode:
If DUTY = 0:
Period(I2C) = 3* CCR * t_{MASTER}
t_{high} = CCR * t_{MASTER}
t_{low} = 2 * CCR * t_{MASTER}
If DUTY = 1: (to reach 400 kHz)
Period(I2C) = 25 * CCR * t_{MASTER}
t_{high} = 9 * CCR * t_{MASTER}
t_{low} = 16 * CCR * t_{MASTER}

Note: *t_{CK} = 1/ f_{MASTER}. f_{MASTER} is the input clock to the peripheral configured using clock control register.*
The minimum allowed value is 04h, except in FAST DUTY mode where the minimum allowed value is 0x01.
t_{high} = $t_{r(SCL)}$ + $t_{w(SCLH)}$. See device datasheet for the definitions of parameters.
t_{low} = $t_{f(SCL)}$ + $t_{w(SCLL)}$. See device datasheet for the definitions of parameters.
I2C communication speed, f_{SCL} = 1/(t_{high} + t_{low})
The real frequency may differ due to the analog noise filter input delay.

21.7.12 Clock control register high (I2C_CCRH)

Address offset: 0x0C
Reset value: 0x00

7	6	5	4	3	2	1	0
F/S	DUTY	Reserved		CCR[11:8]			
rw	rw	r		rw			

Bit 7 **F/S**: I2C master mode selection
0: Standard mode I2C
1: Fast mode I2C

Bit 6 **DUTY**: Fast mode duty cycle
0: Fast mode t_{low}/t_{high} = 2
1: Fast mode t_{low}/t_{high} = 16/9 (see CCR)

Bits 5:4 Reserved

Bits 3:0 **CCR[11:8]**: Clock control register in Fast/Standard mode (Master mode)
Controls the SCLH clock in master mode.
– Standard mode:
Period(I2C) = 2 * CCR * t_{MASTER}
t_{high} = CCR * t_{MASTER}
t_{low} = CCR * t_{MASTER}
– Fast mode:
If DUTY = 0:
Period(I2C) = 3 * CCR * t_{MASTER}
t_{high} = CCR * t_{MASTER}
t_{low} = 2 * CCR * t_{MASTER}
If DUTY = 1: (to reach 400 kHz)
Period(I2C) = 25 * CCR * t_{MASTER}
t_{high} = 9 * CCR * t_{MASTER}
t_{low} = 16 * CCR * t_{MASTER}
For instance: in standard mode, to generate a 100 kHz SCL frequency:
If FREQR = 08, t_{MASTER} = 125 ns so CCR must be programmed with 0x28

If FREQR = 08, t_{MASTER} = 125 ns so CCR must be programmed with 0x28
(0x28 <=> 40 x 125 ns = 5000 ns.)

Note: $t_{high} = t_{r(SCL)} + t_{w(SCLH)}$. See device datasheet for the definitions of parameters
$t_{low} = t_{f(SCL)} + t_{w(SCLL)}$. See device datasheet for the definitions of parameters
The real frequency may differ due to the analog noise filter input delay.

Note: The CCR registers must be configured only when the I²C is disabled (PE=0).
f_{MASTER} = multiple of 10 MHz is required to generate Fast clock at 400 kHz. $f_{MASTER} \geq 1$ MHz is required to generate Standard clock at 100 kHz.

(/img/stm8/uart/uart1_15.png)

where the CCR value is:

$$Period_{I2C} = 2 * CCR * T_{MASTER}$$

and therefore:

$$CCR = \frac{P\text{It is}riOd_{I2C}}{2 * T_{MASTAND R}}$$

For example, for the I $_2$ C bus frequency = 100 kHz, and the f $_{MASTER}$ frequency = 16 MHz, we have the value CCR = 80:

$$CCR = \frac{\frac{1}{10^5}}{2 * \frac{1}{16 * 10^6}} = \frac{10^{-5}}{2 / 16 * 10^{-6}} = \frac{16 * 10}{2} = 80$$

It is, of course, easier to calculate CCR as the ratio of the f $_{MASTER}$ frequency to the I ^{2} C bus frequency:

$$CCR = \frac{f_{MASTAND R}}{2 * f_{I2C}}$$

In addition to the CCR value, the I2C_CCRH register contains an F/S flag that must be cleared to zero for the standard 100 kHz mode.

3. The third register, which is needed to configure the I2C interface, sets the 7-bit or 10-bit I2C address mode : For the standard 7-bit mode, the ADDMODE flag should be cleared and ADDCONF set.

21.7.5 Own address register MSB (I2C_OARH)

Address offset: 0x04
Reset value: 0x00

7	6	5	4	3	2	1	0
ADDMODE	ADDCONF	Reserved			ADD[9:8]		Reserved
rw	rw	r			rw		r

Bit 7 **ADDMODE** Addressing mode (Slave mode)
0: 7-bit slave address (10-bit address not acknowledged)
1: 10-bit slave address (7-bit address not acknowledged)

Bit 6 **ADDCONF** Address mode configuration
This bit must set by software (must always be written as ‘1’).

Bits 5:3 Reserved

Bits 2:1 **ADD[9:8]** Interface address
10-bit addressing mode: bits 9:8 of address.

Bit 0 Reserved

(/img/stm8/uart/uart1_16.png)

4. The last register that is needed for I ^{2}C configuration is I2C_CR1: Here only the PE flag is of interest, which enables/disables the module. **All configuration must occur with the PE flag reset!**

21.7.1 Control register 1 (I2C_CR1)

Address offset: 0x00
Reset value: 0x00

7	6	5	4	3	2	1	0
NOSTRETCH	ENGC	Reserved					PE
rw	rw	r					rw

Bit 7 **NOSTRETCH**: Clock stretching disable (Slave mode)
This bit is used to disable clock stretching in slave mode when ADDR or BTF flag is set, until it is reset by software.
0: Clock stretching enabled
1: Clock stretching disabled

Bit 6 **ENGC**: General call enable
0: General call disabled. Address 0x00 is NACKed.
1: General call enabled. Address 0x00 is ACKed.

Bits 5:1 Reserved

Bit 0 **PE**: Peripheral enable
0: Peripheral disable
1: Peripheral enable: the corresponding I/Os are selected as alternate functions.
Note: If this bit is reset while a communication is on going, the peripheral is disabled at the end of the current communication, when back to IDLE state.
All bit resets due to PE=0 occur at the end of the communication.

(/img/stm8/uart/uart1_17.png)

4. I ^{2}C in master mode, using RTC DS1307/DS3231 as an example. Address and data transfer function

The function for transferring an I ^{2}C address and writing one number turned out to be like this (the function parameters are read from the stack):

```
_init_i2c:
    ;---------- Begin  I2C routine ------------------
    ;   Send Address
    bset I2C_CR2,#2         ; set ACK bit
    bset I2C_CR2,#0         ; START
wait_start_tx:              ; wait SB in I2C_SR1
    btjf I2C_SR1, #0, wait_start_tx
    ld a,I2C_SR1            ; Clear SB bit
    ld a, (03,sp)
    ld I2C_DR,a             ; I2C address
wait_adr_tx:
    btjf I2C_SR1,#1, wait_adr_tx
    ld a,I2C_SR1            ; clear ADDR bit
    ld a,I2C_SR3            ; clear ADDR bit
    ld a,(04,sp)
    ld I2C_DR,a             ; DS1307 set address =0
wait_zero_tx:               ; wait set TXE bit
    btjf I2C_SR1,#7, wait_zero_tx
    bset I2C_CR2,#1         ; STOP
;-------------------------------------------------
    bres I2C_CR2,#7         ; set SWRST
;-------------------------------------------------
    ret
```

This is probably the simplest case, here the slave's response is not analyzed, it is always assumed that it is on the line and responds with an ACK.

First, the START signal is sent, then the device address, then one byte of data. After that, STOP and line reset are sent. The function is suitable for checking the I ^{2}C module. Three cycles provide a guaranteed hang of the microcontroller if something is not working properly.

The important point is that **to clear the status flags you need to read the registers entirely , rather than trying to clear them with a bit operation** .

Registers used in the function:

5. Second configuration register I2C_CR2:

21.7.2 Control register 2 (I2C_CR2)

Address offset: 0x01

Reset value: 0x00

7	6	5	4	3	2	1	0
SWRST	Reserved			POS	ACK	STOP	START
rw	r			rw	rw	rw	rw

Bit 7 **SWRST**: Software reset

When set, the I2C is at reset state. Before resetting this bit, make sure the I2C lines are released and the bus is free.

0: I2C Peripheral not at reset state
1: I2C Peripheral at reset state

Note: This bit can be used in case the BUSY bit is set to '1' when no stop condition has been detected on the bus.

Bits 6::4 Reserved

Bit 3 **POS**: Acknowledge position (for data reception).

This bit is set and cleared by software and cleared by hardware when PE=0.

0: ACK bit controls the (N)ACK of the current byte being received in the shift register.
1: ACK bit controls the (N)ACK of the next byte which will be received in the shift register.

Note: The POS bit is used when the procedure for reception of 2 bytes (see Method 2: transfer sequence diagram for master receiver when N=2) is followed. It must be configured before data reception starts. In this case, to NACK the 2nd byte, the ACK bit must be cleared just after ADDR is cleared.

Note:

Bit 2 **ACK**: Acknowledge enable

This bit is set and cleared by software and cleared by hardware when PE=0.

0: No acknowledge returned
1: Acknowledge returned after a byte is received (matched address or data)

Bit 1 **STOP**: Stop generation

The bit is set and cleared by software, cleared by hardware when a Stop condition is detected, set by hardware when a timeout error is detected.

– In Master mode:
0: No Stop generation.
1: Stop generation after the current byte transfer or after the current Start condition is sent.
– In Slave mode:
0: No Stop generation.
1: Release the SCL and SDA lines after the current byte transfer.

Bit 0 **START**: Start generation

This bit is set and cleared by software and cleared by hardware when start is sent or PE=0.

– In Master mode:
0: No Start generation
1: Repeated start generation
– In Slave mode:
0: No Start generation
1: Start generation when the bus is free

(/img/stm8/uart/uart1_18.png)

here the START flag when set sends the corresponding signal to the line, after which the I ^{2} C module switches to the master mode. Setting this flag in the master mode sends the RESTART signal, this allows you to replace the STOP - START sequences when switching modes for the slave: writing to reading and vice versa. DS1307 does not support the RESTART signal, but other devices, such as LM75, support it. The flag is reset automatically when the START state is set or when the module is disconnected (PE=0).

STOP flag. Setting it sends the corresponding signal to the line, after which the I ^{2} C module is switched to slave mode.

Setting the SWRST flag resets, releases the line.

The POS and ACK flags are used in read mode. ACK - determines whether to send ACK or NACK in response to a received byte (if the ACK flag is set, then send, and if the ACK flag is cleared, then send NACK). POS determines whether to send ACK/NACK with the current or next byte.

6. First flag register I2C_SR1:

21.7.7 Status register 1 (I2C_SR1)

Address offset: 0x07
Reset value: 0x00

7	6	5	4	3	2	1	0
TXE	RXNE	Reserved	STOPF	ADD10	BTF	ADDR	SB
r	r	r	r	r	r	r	r

Bit 7 **TXE**: Data register empty (transmitters) (1)
0: Data register not empty
1: Data register empty
– Set when DR is empty in transmission. TXE is not set during address phase.
– Cleared by software writing to the DR register or by hardware after a start or a stop condition or when PE=0.
Note: TXE cannot be cleared by writing the first data in transmission or by writing a data when the BTF bit is set as in both cases, the DR register is still empty.

Bit 6 **RXNE**: Data register not empty (receivers) (2) (3)
0: Data register empty
1: Data register not empty
– Set when data register is not empty in receiver mode. RXNE is not set during address phase.
– Cleared by software reading or writing the DR register or by hardware when PE=0.
Note: RXE cannot be cleared by reading a data when the BTF bit is set as the DR register is still full in this case.

Bit 5 Reserved

Bit 4 **STOPF**: Stop detection (Slave mode) (4)(5)
0: No Stop condition detected
1: Stop condition detected
– Set by hardware when a Stop condition is detected on the bus by the slave after an acknowledge (if ACK=1).
– Cleared by software reading the SR1 register followed by a write in the CR2 register, or by hardware when PE=0

Bit 3 **ADD10**: 10-bit header sent (Master mode) (6)
0: No ADD10 event occurred.
1: Master has sent first address byte (header).
– Set by hardware when the master has sent the first byte in 10-bit address mode.
– Cleared by software reading the SR1 register followed by a write in the DR register of the second address byte, or by hardware when PE=0.

Bit 2 **BTF**: Byte transfer finished (7)(8)
0: Data byte transfer not done
1: Data byte transfer succeeded
– Set by hardware when NOSTRETCH=0 and:
  – In reception when a new byte is received (including ACK pulse) and DR has not been read yet (RXNE=1).
  – In transmission when a new byte should be sent and DR has not been written yet (TXE=1).
– Cleared by software reading SR1 followed by either a read or write in the DR register or by hardware after a start or a stop condition in transmission or when PE=0.

Bit 1 **ADDR**: Address sent (master mode)/matched (slave mode) (8)(9)
This bit is cleared by software reading SR1 register followed reading SR3, or by hardware when PE=0.
– Address matched (Slave)
0: Address mismatched or not received.
1: Received address matched.
  – Set by hardware as soon as the received slave address matched with the OAR registers content or a general call or a SMBus is recognized. (when enabled depending on configuration).
– Address sent (Master)
0: No end of address transmission
1: End of address transmission
  – For 10-bit addressing, the bit is set after the ACK of the 2nd byte.
  – For 7-bit addressing, the bit is set after the ACK of the byte

– For 7-bit addressing, the bit is set after the ACK of the byte.

Note: ADDR is not set after a NACK reception

Bit 0 **SB**: Start bit (Master mode) (8)

0: No Start condition
1: Start condition generated.
– Set when a Start condition generated.
– Cleared by software by reading the SR1 register followed by writing the DR register, or by hardware when PE=0

1. The interrupt will be generated when DR is copied into shift register after an ACK pulse. If a NACK is received, copy is not done and TXE is not set.
2. The interrupt will be generated when Shift register is copied into DR after an ACK pulse.
3. RXNE is not set in case of ARLO event.
4. The STOPF bit is not set after a NACK reception.
5. It is recommended to perform the complete clearing sequence (READ SR1 then WRITE CR2) after STOPF is set. Refer to *Figure 103: Transfer sequence diagram for slave receiver on page 293*
6. The ADD10 bit is not set after a NACK reception.
7. The BTF bit is not set after a NACK reception, or in case of an ARLO event.
8. Due to timing constraints, when in standard mode if CCR is less than 9 (i.e. with peripheral clock below 2 MHz) with $f_{MASTER} = f_{CPU}$ and the event interrupt disabled, the following procedure must be followed:
modify the reset sequence in order to insert at least 5 cycles between each operations in the flag clearing sequence. For example, when $f_{MASTER} = f_{CPU} = 1$ MHz, use the following sequence to poll the SB bit:
```
_label_wait: BTJF I2C_SR1,SB,_label_wait
NOP ;
NOP;
NOP ;
NOP
NOP
LD I2C_DR, A ; once executed, the SB bit is then cleared.
```
9. In slave mode, it is recommended to perform the complete clearing sequence (READ SR1 then READ SR3) after ADDR is set. Refer to *Figure 103: Transfer sequence diagram for slave receiver on page 293*.

(/img/stm8/uart/uart1_19.png)

Here, the SB flag is set when the START signal is on the line. The flag is cleared by software by reading the I2C_SR1 register. The flag is cleared by hardware when the module is disconnected (PE=0).

The ADDR flag is set when transmitting an I2C address, including receiving an ACK response. **The ADDR flag is NOT set when not receiving an ACK response from the slave!** The flag is reset by reading the I2C_SR1 register followed by reading I2C_SR3. The flag is cleared by hardware when the module is disconnected (PE=0).

BTF flag - completion of exchange operation. Set in read mode, when a new byte has been received, but not yet read from the DR data register. Also set during transmission, when a byte has been sent, and the DR data register has not yet been written with a new value. The flag is cleared: by software when reading the SR1 register, by hardware when writing or reading the DR data register, or when the module is disconnected (PE=0).

STOPF flag - set by hardware when a stop signal is detected on the line. The flag is cleared by software by reading the I2C_SR1 register followed by reading I2C_SR2. The flag is cleared by hardware when the module is disconnected (PE=0).

RxNE flag (data register not empty) - is set by hardware when receiving data in the DR register. The flag is cleared by software by reading or writing to the DR register. The flag is cleared by hardware when the module is disconnected (PE=0).

TxE flag (data register is empty) - set by hardware when a byte is moved from the DR data register to the shift register for transmission to the line. The flag is cleared by software by writing to the DR register. The flag is cleared by hardware after START or STOP signals on the line or when the module is disconnected (PE=0).

5. I 2 C in master mode, using RTC DS1307/DS3231 as an example. Data reading function

A powerful feature of STM8 is that data from the slave can be read in portions of one, two, three bytes or some array. Often, when working via I2C, it is necessary to read only one or two bytes of the device and here we have the opportunity to use a simplified reading algorithm without cycles.

A distinctive feature of the I 2 C module is that **the ACK, NACK, STOP signals are transmitted in a "packet" together with the read byte, and they must be configured BEFORE (!) reading this byte from the DR register** .

Since the master completes the reading session by transmitting NACK and STOP signals, the order of setting flags to generate these signals is what differentiates the algorithms for completing a reading session in one, two or three bytes.

Ending a session in one byte seemed to me to be the most universal. With its help, you can read both a single byte and an array of bytes. The function for reading an array of seven bytes is presented below:

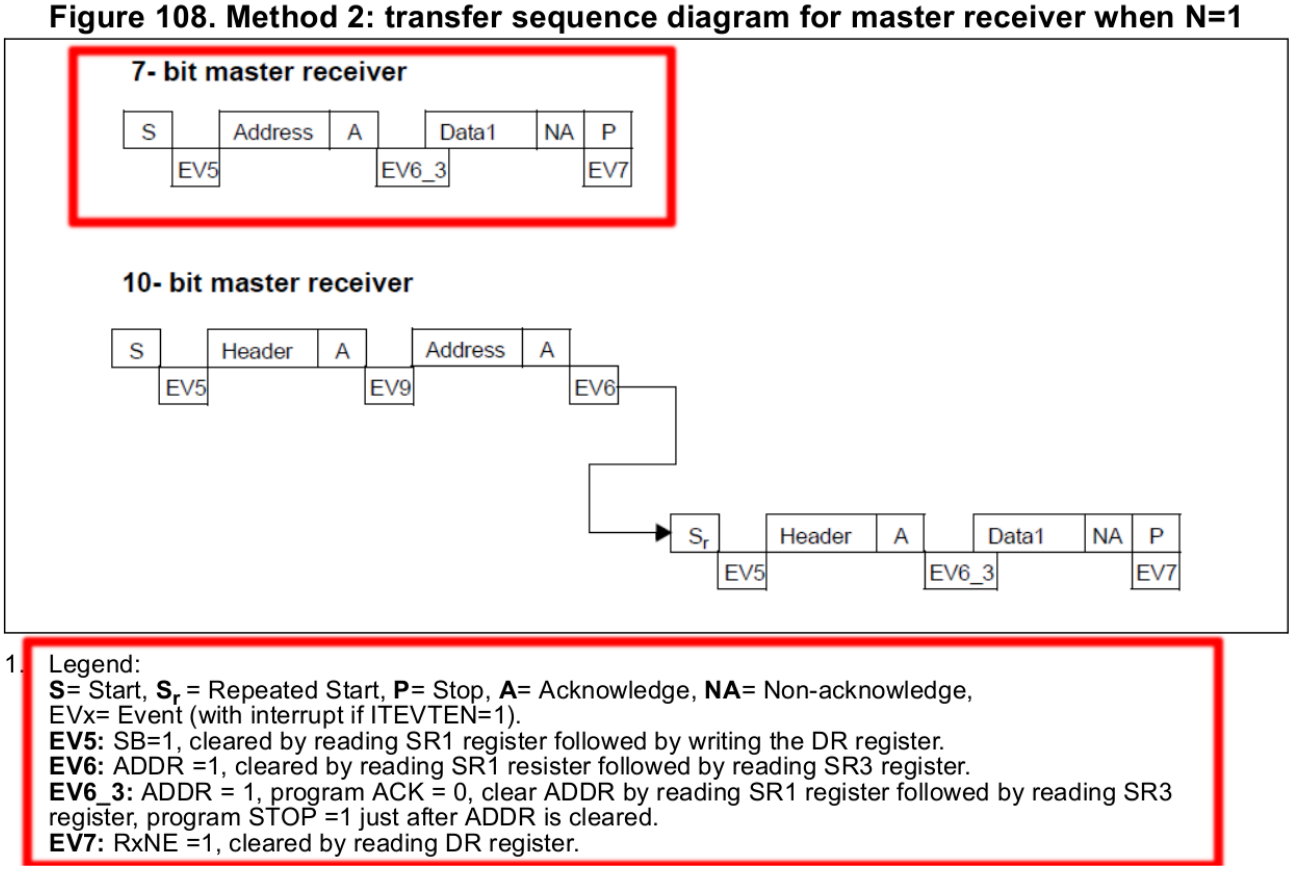
```
_read_i2c:
;-------------------------------------------------
    bset I2C_CR2,#2         ; set ACK bit
    bset I2C_CR2,#0         ; START
wait_start_rx:              ; wait SB in I2C_SR1
    btjf I2C_SR1, #0, wait_start_rx
    ld a,I2C_SR1            ; Clear SB bit
    ld a,(03,sp)
    inc a
    ld  I2C_DR, a           ; DS1307 address RAED Mode
wait_adr_rx:
    btjf I2C_SR1,#1, wait_adr_rx
    ; --------- READ 7 BYTES --------------------
    bset I2C_CR2,#2         ; send ACK
    ld a,I2C_SR1            ; clear ADDR bit
    ld a,I2C_SR3            ; clear ADDR bit
    clrw y
    ldw x,(04,sp)
    ; --------- READ LOOP ----------------------
wait_read:
    btjf I2C_SR1,#6,wait_read
    ld a,I2C_DR
    ld (x),a
    incw x
    incw y
    cpw y,#06
    jrne wait_read
    ; get last byte
    bres I2C_CR2,#2         ; NACK
    bset I2C_CR2,#1         ; STOP
wait_last:                    ; wait RXNE bit
    btjf I2C_SR1,#6, wait_last
    ld a,I2C_DR             ; get data from DS1307 and clear RXNE bit
    ld (x), a               ; save seconds in RAM

    bres I2C_CR2 ,# 7          ; set SWRST

    right
```

Here the device address and the array pointer are passed through the stack.

If we remove the six-byte read cycle from the function, we are left with a single-byte read function. As you can see, the STOP flag is set and the ACK flag is reset before the last byte is read from the DR register.

The order of reading one byte in the documentation is presented by the following diagram:

Figure 108. Method 2: transfer sequence diagram for master receiver when N=1



1. Legend:
S= Start, **S_r** = Repeated Start, **P**= Stop, **A**= Acknowledge, **NA**= Non-acknowledge,
EVx= Event (with interrupt if ITEVTEN=1).
EV5: SB=1, cleared by reading SR1 register followed by writing the DR register.
EV6: ADDR =1, cleared by reading SR1 resister followed by reading SR3 register.
EV6_3: ADDR = 1, program ACK = 0, clear ADDR by reading SR1 register followed by reading SR3 register, program STOP =1 just after ADDR is cleared.
EV7: RxNE =1, cleared by reading DR register.

(/img/stm8/uart/uart1_20.png)

One, two or three bytes can be read without using cycles. For example, to read the current time from DS3231, you need to read three bytes. Then the read function will look like this:

```
.globl _read_i2c
_read_i2c:
;------------------------------------------------
    bset I2C_CR2,#2         ; set ACK bit
    bset I2C_CR2,#0         ; START
wait_start_rx:              ; wait SB in I2C_SR1
    btjf I2C_SR1, #0, wait_start_rx
    ld a,I2C_SR1            ; Clear SB bit
    ld a,(03,sp)
    inc a
    ld  I2C_DR, a      ; DS1307 address RAED Mode
    ldw x, (04,sp)
w0:
    btjf I2C_SR1,#1, w0
    ; --------- READ THREE BYTES --------------------
    bset I2C_CR2,#2         ; send ACK
    ld a,I2C_SR1            ; clear ADDR bit
    ld a,I2C_SR3            ; clear ADDR bit
w2:
    btjf I2C_SR1,#2,w2
    bres I2C_CR2,#2         ; NACK
    ld a,I2C_DR
    ld (x),a
    bset I2C_CR2,#1         ; STOP
    ld a,I2C_DR
    incw x
    ld (x),a
w4:
    btjf I2C_SR1,#6,w4
    ld a,I2C_DR
    incw x
    ld (x),a

    bres I2C_CR2 ,# 7          ; set SWRST


    right
```

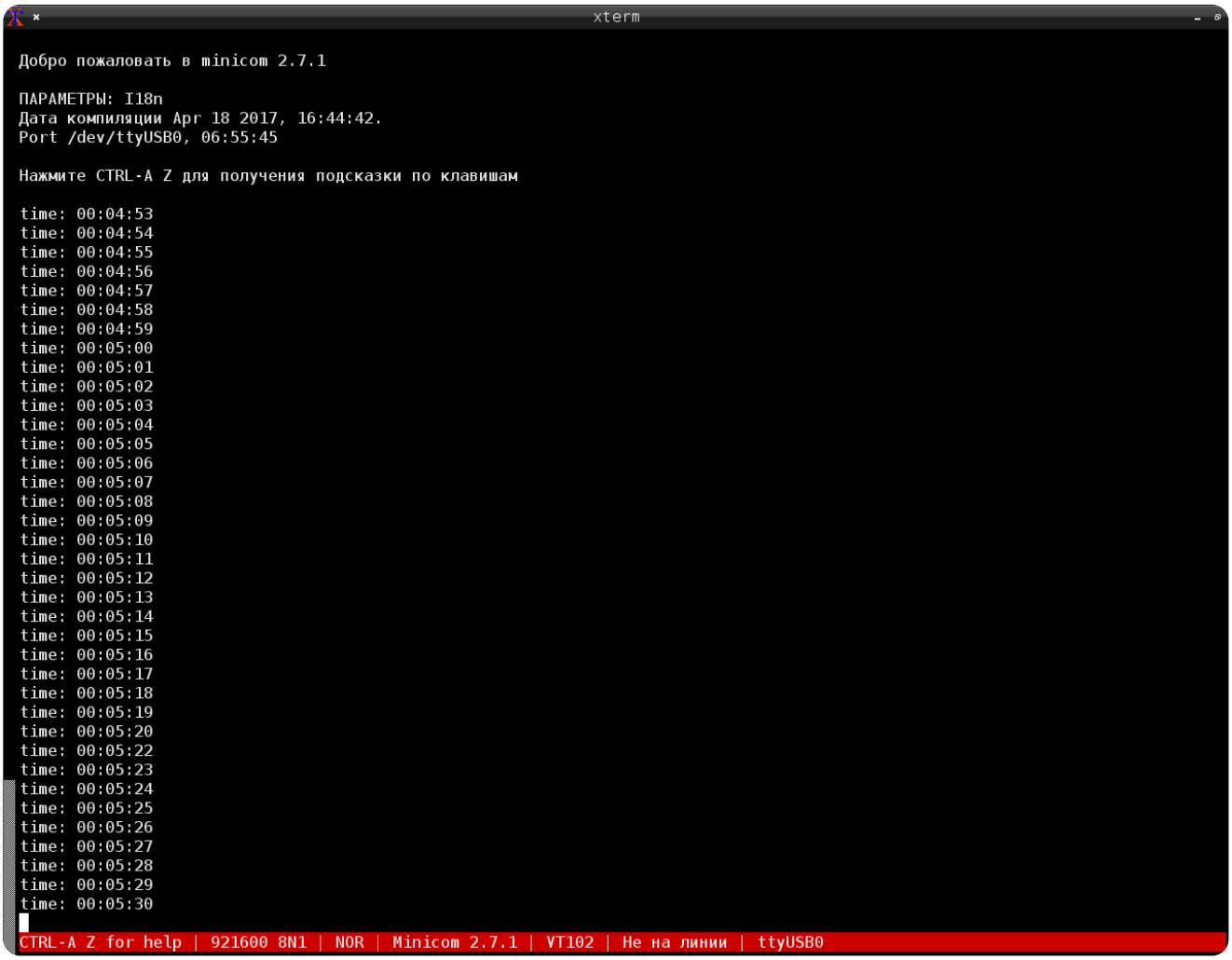
Complete C part of the program for reading data from DS3231:

```
#include <stdint.h>
#include "stm8s103f.h"
#include "uart1.h"

#define LED 3
#define DS1307_ADR  0x68

uint8_t data[7];

extern void delay(uint16_t value);
extern void init_i2c(uint8_t adr, uint8_t value);
extern void read_i2c(uint8_t adr, uint8_t *values);

int main( void ) {
    //----------- Setup Clock ----------------------
    // fCPU =16MHz
    CLK_CKDIVR=0;
    // enable UART and turn off other  peripherals
    CLK_PCKENR1=0;
    CLK_PCKENR2= 0 ;
    clk_pckenr1_bset(#3);   // enable UART1
    clk_pckenr1_bset(#0);   // enable I2C
    clk_pckenr2_bset(#2);   // enable AWU
    //----------- Setup GPIO -----------------------
    // GPIO setup
    pc_ddr_bset(#LED);      //  PC_DDR|=(1<<LED)
    pc_cr1_bset(#LED);      //  PC_CR1|=(1<<LED)
    // ------- Setup AWU & Buzzer ------------------------
    BEEP_CSR=0x1e;      // buuuuuuuuuzzzzz (low tone)
    beep_csr_bset(#5);  // Enable BEEP
    delay(100);
    beep_csr_bres(#5);  // Disable BEEP
    //----------- Setup UART1 ----------------------
    // Clear
    UART1_CR1=0;
    UART1_CR2= 0 ;
    UART1_CR3=0;
    UART1_CR4=0;
    UART1_CR5=0;
    UART1_GTR=0;
    UART1_PSCR=0;
    // setup UART1
    uart1_cr1_bset(#5);     // set UARTD, UART1 disable
    // 9600 Baud Rate, when fMASTER=16MHz
    //UART1_BRR2=0x03
    //UART1_BRR1=0x68
    // 115200 Baud Rate, when fMASTER=16MHz
    //UART1_BRR2=0x0b
    //UART1_BRR1=0x08
    // 230400 Baud Rate, when fMASTER=16MHz
    //UART1_BRR2=0x05;
    //UART1_BRR1=0x04;
    // 460800 Baud Rate, when fMASTER=16MHz
    //UART1_BRR2=0x03;
    //UART1_BRR1=0x02;
    // 921600 Baud Rate, when fMASTER=16MHz
    UART1_BRR2=0x01;
    UART1_BRR1=0x01;
    // Trasmission Enable
    uart1_cr2_bset(#3);      // set TEN, Transmission Enable
    // enable UART1
    uart1_cr1_bres(#5);      // clear UARTD, UART1 enable
    //----------- Setup I2C ------------------------
    i2c_cr1_bres(#0);       // PE=0, disable I2C before setup
    I2C_FREQR= 16;          // peripheral frequence =16MHz
    I2C_CCRH = 0;           // =0
    I2C_CCRL = 80;          // 100kHz for I2C
    i2c_ccrh_bres(#7);      // set standart mode(100кHz)
    i2c_oarh_bres(#7);      // 7-bit address mode
    i2c_oarh_bset(#6);      // see reference manual
    i2c_cr1_bset(#0);       // PE=1, enable I2C
    //------------- End Setup ---------------------

    // main loop
    for(;;)
    {
        init_i2c((DS1307_ADR<<1), 0x0);
        read_i2c((DS1307_ADR<<1), data);
        pc_odr_bcpl(#LED);
        uart1_print_string("time: ");
        uart1_print_bcd(data[2]);
        uart1_send_char(':');
        uart1_print_bcd(data[1]);
        uart1_send_char(':');
        uart1_print_bcd(data[0]);
        uart1_send_char('\n');
        delay(800);
    }
}
```

I would like to draw attention to the fact that the LED for indicating the working cycle was moved from PB5 to PC3, since pin PB5 is occupied by the I ^{2} C line.

The text of the init_i2c() and read_i2c() functions was presented above.

The result of work in the minicom program window:



(/img/stm8/uart/uart1_22.png)

You can download the full source code with assembly files and compiled firmware using this link (https://www.dropbox.com/s/864z7vz7e4oxkwa/stm8_uart_adc_i2c.zip?dl=1) .

(/)

Михаил Батон
год назад

Насчет повисаний АЦП. Бит enable ADC, он же start adc его нужно включать два раза подряд (24.5.1, стр. 431).

1 o Ответить Поделиться ›

Karabas Barabas
6 лет назад

Прошу прощения, хотел задать вопрос. А почему eclipse не любите ? Я например живу с ним с 2001-го года, со 2-й ветки. Умею делать с ним абсолютно всё, включая написание плагинов под свои нужды или под заказ. Единственно что сильно огорчает, это то что начиная с 4.2 он становится всё хуже и хуже :(((Последняя 4.7.2 тормозит и отжирает памяти похуже чем продукция от JetBrains. Тем не менее заменить его пока что НЕЧЕМ. От слова совсем. Просто на примере моего проекта на stm8s, о котором упоминал. Код под stm8s написан на C. Прошивка FPGA (тестирующее окружение) на верилоге. Все тесты на java. Всё это открыто в одной среде. Дико удобно ! Не знаю чем это можно заменить... Единственно надеюсь что разработчики всё-таки поймут, что полезли куда-то не туда, и положение начнет улучшаться. Проект по-моему совершенно гениальный. Было бы обидно, если он умрет. Для меня это пожалуй вообще первая любовь в мире компьютерных программ. Она же наверно и последняя. Ибо ни до ни после не видал систем, АКТИВНО ПОМОГАЮЩИХ (!!!) в изучении своего собственного устройства.

o o Ответить Поделиться ›

flanker Модератор → Karabas Barabas
6 лет назад edited

>почему eclipse не любите ?
Потому что не люблю java в принципе, как впрочем недолюбливаю все интерпретируемые языки программирования(JS, Python, Perl не в счет, у них есть есть сильные стороны которые все перевешивают). Не люблю за более низкую скорость по сравнению с компилируемыми языками. Да, в курсе в java team многое сделали чтобы доказать что это миф, но... чисто субъективно сравнивая отзывчивость Qt Creator и Eclipse, у последнего нет шансов. Почитайте это: http://www.count-zero.ru/20... ту часть, что относится к Eclipse, у меня тогда он валился каждые две минуты. Сборка Eclipse которая мне действительно понравилась, и заставила изменить свое мнение, это: Code Composer Studio. Но все же, если у меня будет выбор, то Eclipse(да и любую другую IDE на Java) я выберу в последнюю очередь.

Потом, мне еще не нравится концепция, когда все настройки проекта хранятся в куче вкладок. Мне гораздо проще написать конфиг CMake проекта и открыть его в Qt Creator ничего более не настраивая. Гениально, как я считаю.

зы а за отзыв вам спасибо)

o o Ответить Поделиться ›

Karabas Barabas
6 лет назад

Огромное спасибо ! Потрясающий блог про железки ! Только с помощью Вашего сайта и небольшой платки с FPGA поднял довольно сложный проект на stm8s. Причем винду с IAR EWB пришлось запускать лишь однажды, чтобы разобраться с pwm на TIM1, уж больно он собака с прибабахами. Всё остальное без SPL только по Вашим описаниям. Короче полный респект и уважуха :))

o o Ответить Поделиться ›

Andrey Kalinin
6 лет назад

Отличные статьи! Все последовательно, точно и нереально интересно! Огромное спасибо count-zero !

o o Ответить Поделиться ›