home

Setting up I2C on the STM8 microcontroller

Website:
STM8
Today I would like to briefly talk about the I2C module on the STM8S003F microcontroller and share my developments in this area.

To be honest, I had to tinker with the I2C module for a long time - it just wouldn't work the way it should until I read the errata and fixed some other small but insidious errors that migrated into my program from the example provided on the official STMicroelectronics website.

In this article I will show how to set up I2C in master mode without interrupts. At the very end of the article you can download a ready-made hardware I2C driver for STM8. In the next article we will try to connect a DS1307 real-time clock and output the time via UART. Perhaps in the future there will be an example on interrupts, or maybe someone wants to share their developments - you are welcome.

For those who want to understand the principle of operation of the I2C interface, I recommend reading a short and at the same time useful article: **IIC (I2C) Interface Bus**

I also recommend reading a useful note: **How I defeated I2C (STM8L)**

You will also need to read the article on **setting up the UART module** and familiarize yourself with the **STM8 S003** datasheets and **Reference manual** .

Let's consider the main capabilities of the I2C module:

- Support Master and Slave mode
- Generation and definition of 7- and 10-bit addresses
- Support different transmission speeds:
    - Standard mode 100 kHz
    - High speed mode 400 kHz

As usual, we start with initialization .

1. Set the clock frequency of the I2C module. This is the Fmaster frequency, which is taken before the CPUDIV divider. If you read the previous article on setting up the UART module, you know that by default the RC generator (16 MHz) is selected as the clock source, with the frequency divided by 8. That is, by default Fmaster = 2 MHz. The Fmaster value is written to the **I2C_FREQR** register in MHz. The values that can be written to this register are in the range of 1 ... 24 MHz.

2. Next, we need to decide in which mode we will work. DS1307 works only in standard mode (100 kHz), so we select it. Standard mode is selected by resetting the **F/ S** bit in the **I2C_CCRH register.**

3. Set up **the I2C_CCR L** and **I2C_CCRH** clock control registers . These registers determine the duration of the SCL clock pulse and the duration of the pause.

I2C_CCR L

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

I2C_CCRH

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

The CCR coefficient is calculated using different formulas (they are described in the reference manual), depending on the selected operating mode. For the standard mode, it is calculated using the formula:

CCR= Period_I2C/(2*Tmaster), where

Period_I2C - the period of SCL pulses on the I2C bus, in standard mode the minimum period is 1/100 kHz,

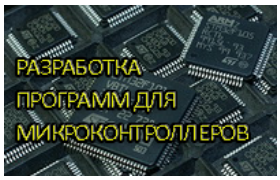
Tmaster - the period of the peripheral clock frequency, Tmaster= 1/Fmaster= 1/2 000 000 Hz.

Thus, the formula takes the form:

CCR= (Fmaster* 1,000,000)/(2 * Fi2c) , where

Sections

Microcontrollers
Scheme
Forum
For Beginners
Directory
Programs
Calculations
Technologies
Notes



Latest articles

Building and installing the QtMqtt module (QMqttClient) for Qt 5.15.2
Debugging Raspberry Pi Pico RP2040 Code in Eclipse
Working with Raspberry Pi Pico RP2040 in Eclipse on Windows
STM32F103C8T6+W5500 Debug Board
Debug board ZQ_STM32F107_Mini
Debug board miniSTM32F103R_-EK
Developing C Programs for Orange Pi Zero in Eclipse
Blinking an LED on an Orange Pi Zero
Programming Milandr microcontrollers in CooCox IDE 1.7

Fmaster - peripheral frequency in MHz,

Fi2c – I2C bus frequency.

The DUTY bit allows you to select the duty cycle for high speed mode.

4. Program the maximum rise time of the SCL signal. According to the specification, the rise time of the SCL signal in standard mode should not exceed 1000 ns. The rise time is set in the **TRISE** register .

The value of the register is calculated using the formula:

TRISE= (1000 ns/Tmaster) + 1 .

If the frequency Fmaster=2MHz, then the period is 500ns, then the register value is:

TRISE = (1000ns/500ns) + 1 = 3

5. Turn on the I2C module by setting the **PE** bit =1 in the **I2C_CR1 register.**

6. Set the bit that enables confirmation of the received byte **AC K** =1 in the **I2C_CR2 register**

I will briefly describe *the status and control registers* that we will have to work with further.

I2C_CR2

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

- **SWRST** – software reset of the I2C module
- **POS** – this bit is set only when two bytes need to be received (N=2). It must be set before data is received. In this case, the NACK pulse will be generated after the second byte is received, and the ACK can be reset immediately after sending the address.
- **ACK** – allows sending acknowledgements from the master after receiving each byte
- **STOP** – instructs the master to generate a stop message after receiving the current byte
- **START** – instructs the master to generate a start message

I2C_SR 1

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

- **TXE** – DR data register is empty
- **RXNE** – the DR data register contains the received byte
- **STOPF** – a stop parcel was detected on the bus
- **ADD10** – the master sent the first byte of the 10-bit address
- **BTF** – is set when a byte has arrived in the receiving register DR, but has not been read, and a new byte has already arrived in the shift register
- **ADDR** – the master has transmitted the address
- **SB** – the master generated a start-parcel

I2C_SR 2

7	6	5	4	3	2	1	0
Reserved		WUFH	Reserved	OVR	AF	ARLO	BERR
		rc_w0		rc_w0	rc_w0	rc_w0	rc_w0

- **WUFH** – exit from sleep mode
- **OVR** – Data Register Overflow Error
- **AF** – Error acknowledging byte sent by master
- **ARLO** – Arbitration Loss (on a bus with more than one master)
- **BERR** – bus error (for example when an inappropriate start or stop is sent)

I2C_SR3

7	6	5	4	3	2	1	0
DUALF	Reserved		GENCALL	Reserved	TRA	BUSY	MSL
r			r		r	r	r

- TRA – if data received = 0, if transmitted = 1
- BUSY – signals that the bus is busy
- MSL – shows what mode the I2C module is in. 1 – master, 0 – slave

There is also an interrupt setup register I2C_ITR, which we will not configure in this example. Also for the slave there are registers in which the device's own I2C address is written - **I2C_OARL, I2C_OAR H** .

Below I have translated some information from the reference manual for you. If I made a mistake somewhere, please correct me.

Master mode

In master mode, the I2C interface initiates data transfer and generates a clock signal. Each transfer begins with a START condition and ends with a STOP condition. Master mode is enabled immediately after START is generated.

The input frequency must be no less than:

- 1 MHz in standard mode
- 4 MHz in fast mode

Starting condition (starting premise)

Setting the START bit results in the generation of a start packet and switches to master mode, provided that BUSY= 0.

Note: In master mode, setting the START bit generates a restart message at the end of the current byte being transmitted.

Immediately after the start frame, the SB bit is set and an interrupt is generated if it is enabled ITEVTEN= 1.

Next, the master waits for the SR1 register to be read, after which the slave device address (slaveaddress) is written into the DR register.

Transferring slave address

The slave address is transferred to the SDA line from the shift register.

● In 10-bit addressing mode, sending is accompanied by the following events:

The ADD10 bit is set by hardware and an interrupt is generated if enabled in ITEVTEN.

The master then waits for the SR1 register to be read, after which the second part of the address is written to DR.

The ADDR bit is set in hardware and an interrupt is generated if enabled in ITEVTEN.

The master then waits for the SR1 register and the SR3 register to be read, allowing the ADDR bit to be cleared and transmission to continue.

● In 7-bit addressing mode, only one byte is sent.

Immediately after the address byte is sent, the ADDR bit is set in hardware and an interrupt is generated if enabled in ITEVTEN.

The master then waits for the SR1 register and the SR3 register to be read.

The master can decide to go into transmit or receive mode according to bit 0 of the slave address.

If the zero bit = 0, then transmit, if = 1, then receive.

● In 10-bit addressing mode:

– To enter the transmission mode, the master sends a header (11110xx0) and a slave address (where xx is the two most significant bits of the address).

– To enter the receive mode, the master sends the header (11110xx0) slave address. Then a repeated restart-package is sent, then the header (11110xx1).

The TRA bit indicates what mode the master is in: receiving or transmitting.

Master in transmitter mode

After transmitting the address and clearing the ADDR bit, the master sends the byte from the DR register to the SDA line via the internal shift register.

The master waits until the first byte of data is written to DR (event EV8_1).

When the confirmation signal (pulse) is received:

● The TXE bit is set by hardware and an interrupt is generated if enabled in ITEVTEN, and the ITBUFEN bit is set.

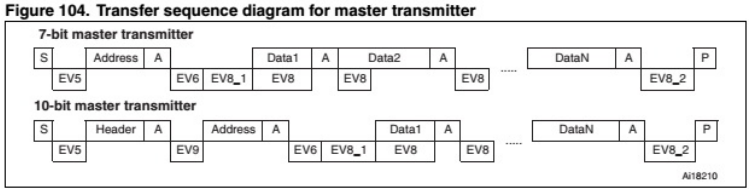
● If TXE is asserted and a data byte has not been written to DR before the end of the next data transfer, the BTF bit is set and the interface waits until it is cleared, which is done by reading SR1 and writing to DR, SCL is held low.

Completion of communication (transmission)

After writing the last byte to DR, the STOP bit is set, which causes the generation of a STOP message (EV8_2 event). The interface automatically switches to slave mode. (MSL bit is reset).

Note : Stop message must be programmed during event EV8_2 when TXE or BTF is set.

Figure 104. Transfer sequence diagram for master transmitter



S= Start,

Sr= Restart,

P= Stop,

A= Confirmation,

NA= non-confirmation,

EVx= event

EV 5 :SB=1 is cleared by reading register SR1, then the address is written to DR.

EV 6 :ADDR=1, cleared by reading register SR1 and then reading SR3.

EV 8_1 :TXE=1, shift register is empty, data register is empty, write data to DR.

EV 8 :TXE=1, shift register is not empty, data register DR is empty, cleared by writing to DR.

EV 8_2 :TXE=1, BTF= 1, STOP request is programmed. TXE and BTF are cleared by hardware after generation of stop message.

EV 9 :ADD10=1, cleared by reading register SR1, then writes to register DR.

EV 8 event must be executed before the end of the current byte transmission. Otherwise, it is recommended to use BTF instead of TXE with slow communication.

Master in receiver mode

After transmitting the address and clearing the ADDR bit, the I2C interface enters the receiver mode. In this mode, the interface receives bytes from the SDA line into the DR register via an internal shift register.

After each byte, the interface generates a sequence:

● Acknowledgment pulse if the ACK bit is set

● Set the RXNE bit and generate an interrupt if the ITEVTEN and ITBUFEN bits are set.

If the RXNE bit is set and data has not been read from DR before the next byte has been received, the BTF bit is set by hardware and the interface waits for this bit to be cleared by reading I2C_SR1 and I2C_DR, SCL is held low.

Terminating the connection

Method 1 : *This method is suitable if I2C uses interrupts that have the highest priority in the application.*

The master sends a NACK at the end of the last byte from the slave.

After receiving this NACK, the slave releases the SCL and SDA lines. The master can then send a Stop or Restart message.

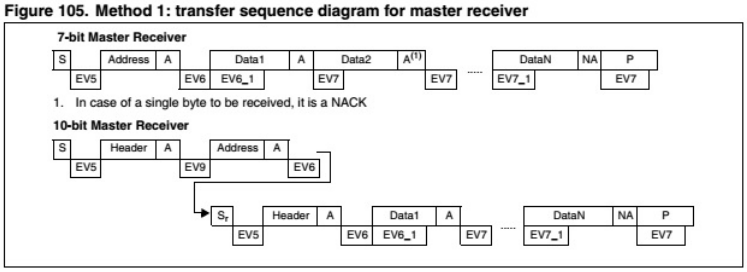
● In case of NACK pulse generation after the last received byte, the ACK bit must be cleared exactly after reading the penultimate byte (after the penultimate RXNE event).

● In case of Stop or Restart generation, the application must set the STOP/START bit immediately after reading the penultimate byte (after the penultimate RXNE event).

● If one byte is received, the acknowledgement is deactivated and a STOP is generated after the EV6 event (in EV6 immediately after ADDR is cleared).

After generating a STOP message, the interface automatically switches to slave mode. (MSL= 0).

Figure 105. Method 1: transfer sequence diagram for master receiver



EV 5 *: SB =1, cleared by reading SR 1 followed by writing to DR .*

EV 6 *: ADDR =1, cleared by reading SR 1 and then reading SR 3. In 10-bit master-receiver mode, this sequence must follow writing START = 1 to CR 2 .*

EV 6_1 *: no flags for this event, used for single byte receive only. Programmable ACK =0 and STOP =1 after ADDR is cleared .*

EV 7 *: RXNE =1, cleared by reading DR .*

EV 7_1 *: RXNE =1, cleared by reading DR , programmed ACK =0 and STOP request*

EV 9 *: ADD 10=1, cleared by reading SR 1 followed by writing to DR .*

1.If the DR register is full, the next data is received after ***the EV 7*** event is cleared .

2. EV5, EV6, and EV9 events hold SCL in a low-level state until the end of the corresponding sequence of actions in the application.

3. EV7 program sequence must be completed before the end of the current byte transmission. Otherwise, it is recommended to use BTF instead of RXNE, thereby slowing down the communication speed.

4. The EV6_1 or EV7_1 sequence must be completed before the ACK pulse of the current byte.

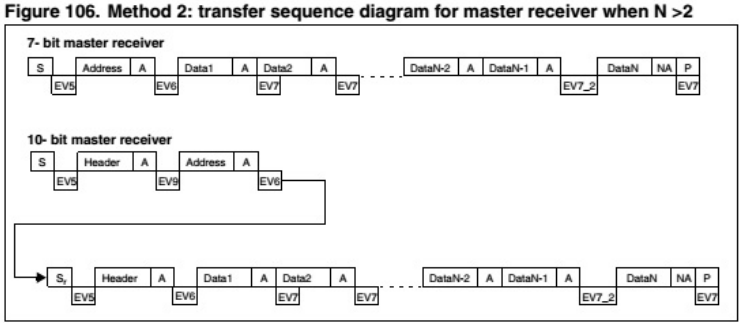
Method 2 : *This method is suitable for cases where interrupts are used that are not the highest priority in the application and when I2C is used in polling mode.*

● DataN_2 is not read, so after DataN_1 the connection slows down (both RxNE and BTF bits are set).

● Next, the ACK bit must be cleared before reading DataN-2 from DR to ensure that this bit has been cleared before the DataN acknowledge pulse.

● After this, immediately after reading DataN_2, the application must set the STOP/START bit and read DataN_1. After setting RXNE, DataN is read.

Figure 106. Method 2: transfer sequence diagram for master receiver when N >2



EV 5 : SB=1, *cleared by reading* SR1 *followed by writing to* DR.

EV 6 :ADDR1, *cleared by reading* SR1 *followed by reading* SR3.

In 10-bit master-receiver mode, this sequence must follow after writing START = 1 bit to CR 2 .

EV 7 : RxNE=1, *cleared by reading* DR.

EV 7_2 : BTF= 1, DataN-2 in DR, and DataN-1 in shift register, ACK= 0 is programmed, DataN-2 data is read from DR. STOP= 1 is set, DataN-1 is read.

EV 9 : ADD10= 1, *cleared by reading* SR1 *followed by writing to* DR.

When there are 3 bytes left to read (N>2) :

● RxNE= 1 => do nothing (DataN-2 is not readable).

● DataN-1 accepted

● BTF= 1 because shift and data register are full: DataN-2 in DR and DataN-1 in shift register => SCL is held low: no further data will be received from the bus.

● Reset ACK= 0

● Read DataN-2 from DR=> This allows DataN to get into the shift register

● DataN received (with NACK sent)

● START/STOP bit is programmable

● DataN-1 data is being read

● Expected to set RxNE= 1

● DataN data is read

The procedure described above is suitable for the case when it is necessary to receive byte N>2.

When you need to receive one (N=1) byte or two (N =2), the processing will be different:

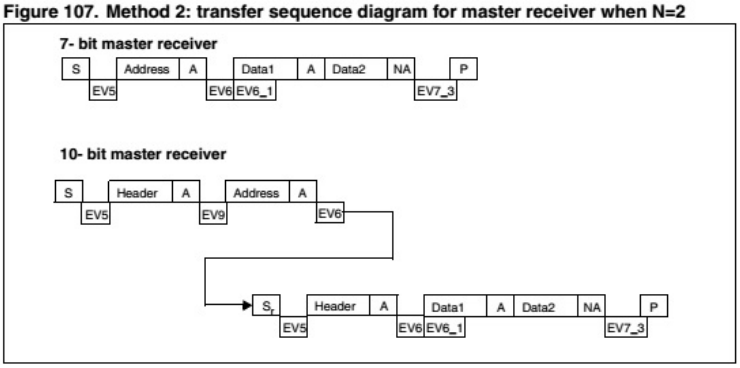
● ***Case N =1***

- Waiting for ADDR to be set, resets the ACK=0 bit.
- Resets ADDR=0
- Set the STOP/START bit.
- We read the data after RxNE is installed.

● ***Case N =2***

- POS=1 and ACK=1 are set
- Expected to set ADDR=1
- ADDR is cleared
- ACK is cleared
- Expected to install BTF=1
- STOP=1 is programmed
- Read DR twice

Figure 107. Method 2: transfer sequence diagram for master receiver when N=2



EV 5 : SB=1, *cleared by reading* SR1 *followed by writing to* DR.

EV 6 :ADDR1, *cleared by reading* SR1 *followed by reading* SR3.

In 10-bit master-receiver mode, this sequence must follow after writing START = 1 bit to CR 2 .

EV 6_1 *: no flags for this event. Acknowledgment should be disabled immediately after EV6 event, after which ADDR is cleared*

EV 7_3 : BTF= 1, STOP=1 is programmed, DR register is read twice immediately after STOP bit is set.

EV 9 : ADD10= 1, *cleared by reading* SR1 *followed by writing to* DR.

EV5, EV6, and EV9 events keep SCL in a low state until the end of the corresponding sequence of actions in the application.

EV 6_1 *sequence must be completed before the ACK pulse is set for the current byte being transmitted.*

Well, finally, **a code example** :

```
001. //Результат выполнения операции с i2c
002. typedef enum {
003.     I2C_SUCCESS = 0,
004.     I2C_TIMEOUT,
005.     I2C_ERROR
006. } t_i2c_status;
007.
008. //Таймаут ожидания события I2C
009. static unsigned long int i2c_timeout;
010.
011. //Задать таймаут в микросекундах
012. #define set_tmo_us(time)\
013.   i2c_timeout = (unsigned long int)(F_MASTER_MHZ * time)
014.
015. //Задать таймаут в миллисекундах
016. #define set_tmo_ms(time)\
017.   i2c_timeout = (unsigned long int)(F_MASTER_MHZ * time * 1000)
018.
019. #define tmo               i2c_timeout--
020.
021. //Ожидание наступления события event
022. //в течении времени timeout в мс
023. #define wait_event(event, timeout) set_tmo_ms(timeout);\
024.                                    while(event && i2c_timeout);\
025.                                    if(!i2c_timeout) return TIMEOUT;
026.
027. //**************************************************************************
028. // Инициализация I2C интерфейса
029. // f_master_hz - частота тактирования периферии Fmaster
030. // f_i2c_hz - скорость передачи данных по I2C
031. //**************************************************************************
032. void i2c_master_init(unsigned long f_master_hz, unsigned long f_i2c_hz){
033.   unsigned long int ccr;
034.
035.   PB_DDR_bit.DDR4 = 0;
036.   PB_DDR_bit.DDR5 = 0;
037.   PB_ODR_bit.ODR5 = 1;  //SDA
038.   PB_ODR_bit.ODR4 = 1;  //SCL
039.
040.   PB_CR1_bit.C14 = 0;
041.   PB_CR1_bit.C15 = 0;
042.
043.   PB_CR2_bit.C24 = 0;
044.   PB_CR2_bit.C25 = 0;
045.
046.   //Частота тактирования периферии MHz
047.   I2C_FREQR_FREQ = 12;
048.   //Отключаем I2C
049.   I2C_CR1_PE = 0;
050.   //В стандартном режиме скорость I2C max = 100 кбит/с
051.   //Выбираем стандартный режим
052.   I2C_CCRH_F_S = 0;
053.   //CCR = Fmaster/2*Fiic= 12MHz/2*100kHz
054.   ccr = f_master_hz/(2*f_i2c_hz);
055.   //Set Maximum Rise Time: 1000ns max in Standard Mode
056.   //= [1000ns/(1/InputClockFrequencyMHz.10e6)]+1
057.   //= InputClockFrequencyMHz+1
058.   I2C_TRISER_TRISE = 12+1;
059.   I2C_CCRL = ccr & 0xFF;
060.   I2C_CCRH_CCR = (ccr >> 8) & 0x0F;
061.   //Включаем I2C
062.   I2C_CR1_PE = 1;
063.   //Разрешаем подтверждение в конце посылки
064.   I2C_CR2_ACK = 1;
065. }
066.
067. //**************************************************************************
068. // Запись регистра slave-устройства
069. //**************************************************************************
070. t_i2c_status i2c_wr_reg(unsigned char address, unsigned char reg_addr,
071.                          char * data, unsigned char length)
072. {
073.
074.   //Ждем освобождения шины I2C
075.   wait_event(I2C_SR3_BUSY, 10);
076.
077.   //Генерация СТАРТ-посылки
078.   I2C_CR2_START = 1;
079.   //Ждем установки бита SB
080.   wait_event(!I2C_SR1_SB, 1);
081.
```

```
082.
083.   //Записываем в регистр данных адрес ведомого устройства
084.   I2C_DR = address & 0xFE;
085.   //Ждем подтверждения передачи адреса
086.   wait_event(!I2C_SR1_ADDR, 1);
087.   //Очистка бита ADDR чтением регистра SR3
088.   I2C_SR3;
089.
090.
091.   //Ждем освобождения регистра данных
092.   wait_event(!I2C_SR1_TXE, 1);
093.   //Отправляем адрес регистра
094.   I2C_DR = reg_addr;
095.
096.   //Отправка данных
097.   while(length--){
098.     //Ждем освобождения регистра данных
099.     wait_event(!I2C_SR1_TXE, 1);
100.    //Отправляем адрес регистра
101.    I2C_DR = *data++;
102.  }
103.
104.  //Ловим момент, когда DR освободился и данные попали в сдвиговый регистр
105.  wait_event(!(I2C_SR1_TXE && I2C_SR1_BTF), 1);
106.
107.  //Посылаем СТОП-посылку
108.  I2C_CR2_STOP = 1;
109.  //Ждем выполнения условия СТОП
110.  wait_event(I2C_CR2_STOP, 1);
111.
112.  return I2C_SUCCESS;
113. }
114.
115. //**************************************************************************
116. // Чтение регистра slave-устройства
117. // Start -> Slave Addr -> Reg. addr -> Restart -> Slave Addr <- data ... ->
       Stop
118. //**************************************************************************
119. t_i2c_status i2c_rd_reg(unsigned char address, unsigned char reg_addr,
120.                          char * data, unsigned char length)
121. {
122.
123.   //Ждем освобождения шины I2C
124.   wait_event(I2C_SR3_BUSY, 10);
125.
126.   //Разрешаем подтверждение в конце посылки
127.   I2C_CR2_ACK = 1;
128.
129.   //Генерация СТАРТ-посылки
130.   I2C_CR2_START = 1;
131.   //Ждем установки бита SB
132.   wait_event(!I2C_SR1_SB, 1);
133.
134.   //Записываем в регистр данных адрес ведомого устройства
135.   I2C_DR = address & 0xFE;
136.   //Ждем подтверждения передачи адреса
137.   wait_event(!I2C_SR1_ADDR, 1);
138.   //Очистка бита ADDR чтением регистра SR3
139.   I2C_SR3;
140.
141.   //Ждем освобождения регистра данных RD
142.   wait_event(!I2C_SR1_TXE, 1);
143.
144.   //Передаем адрес регистра slave-устройства, который хотим прочитать
145.   I2C_DR = reg_addr;
146.   //Ловим момент, когда DR освободился и данные попали в сдвиговый регистр
147.   wait_event(!(I2C_SR1_TXE && I2C_SR1_BTF), 1);
148.
149.   //Генерация СТАРТ-посылки (рестарт)
150.   I2C_CR2_START = 1;
151.   //Ждем установки бита SB
152.   wait_event(!I2C_SR1_SB, 1);
153.
154.   //Записываем в регистр данных адрес ведомого устройства и переходим
155.   //в режим чтения (установкой младшего бита в 1)
156.   I2C_DR = address | 0x01;
157.
158.   //Дальше алгоритм зависит от количества принимаемых байт
159.   //N=1
160.   if(length == 1){
161.     //Запрещаем подтверждение в конце посылки
```

```
    I2C_CR2_ACK = 0;
    //Ждем подтверждения передачи адреса
    wait_event(!I2C_SR1_ADDR, 1);

    //Заплатка из Errata
    __disable_interrupt();
    //Очистка бита ADDR чтением регистра SR3
    I2C_SR3;

    //Устанавлием бит STOP
    I2C_CR2_STOP = 1;
    //Заплатка из Errata
    __enable_interrupt();

    //Ждем прихода данных в RD
    wait_event(!I2C_SR1_RXNE, 1);

    //Читаем принятый байт
    *data = I2C_DR;
  }
  //N=2
  else if(length == 2){
    //Бит который разрешает NACK на следующем принятом байте
    I2C_CR2_POS = 1;
    //Ждем подтверждения передачи адреса
    wait_event(!I2C_SR1_ADDR, 1);
    //Заплатка из Errata
    __disable_interrupt();
    //Очистка бита ADDR чтением регистра SR3
    I2C_SR3;
    //Запрещаем подтверждение в конце посылки
    I2C_CR2_ACK = 0;
    //Заплатка из Errata
    __enable_interrupt();
    //Ждем момента, когда первый байт окажется в DR,
    //а второй в сдвиговом регистре
    wait_event(!I2C_SR1_BTF, 1);

    //Заплатка из Errata
    __disable_interrupt();
    //Устанавлием бит STOP
    I2C_CR2_STOP = 1;
    //Читаем принятые байты
    *data++ = I2C_DR;
    //Заплатка из Errata
    __enable_interrupt();
    *data = I2C_DR;
  }
  //N>2
  else if(length > 2){
    //Ждем подтверждения передачи адреса
    wait_event(!I2C_SR1_ADDR, 1);

    //Заплатка из Errata
    __disable_interrupt();

    //Очистка бита ADDR чтением регистра SR3
    I2C_SR3;

    //Заплатка из Errata
    __enable_interrupt();

    while(length-- > 3 && tmo){
      //Ожидаем появления данных в DR и сдвиговом регистре
      wait_event(!I2C_SR1_BTF, 1);
      //Читаем принятый байт из DR
      *data++ = I2C_DR;
    }
    //Время таймаута вышло
    if(!tmo) return I2C_TIMEOUT;

    //Осталось принять 3 последних байта
    //Ждем, когда в DR окажется N-2 байт, а в сдвиговом регистре
    //окажется N-1 байт
    wait_event(!I2C_SR1_BTF, 1);
    //Запрещаем подтверждение в конце посылки
    I2C_CR2_ACK = 0;
    //Заплатка из Errata
    __disable_interrupt();
    //Читаем N-2 байт из RD, тем самым позволяя принять в сдвиговый
    //регистр байт N, но теперь в конце приема отправится посылка NACK
```

```
    *data++ = I2C_DR;
    //Посылка STOP
    I2C_CR2_STOP = 1;
    //Читаем N-1 байт
    *data++ = I2C_DR;
    //Заплатка из Errata
    __enable_interrupt();
    //Ждем, когда N-й байт попадет в DR из сдвигового регистра
    wait_event(!I2C_SR1_RXNE, 1);
    //Читаем N байт
    *data++ = I2C_DR;
  }

  //Ждем отправки СТОП посылки
  wait_event(I2C_CR2_STOP, 1);
  //Сбрасывает бит POS, если вдруг он был установлен
  I2C_CR2_POS = 0;

  return I2C_SUCCESS;
}
```

I2C Driver for STM8 in IAR

Login or register to post comments

Comments

CCR calculation

PUBLISHED SUN, 12/06/2015 - 19:41 BY ALEX2015

Good day.
Why is the CCR calculation for the standard mode done according to the formula
CCR= Period_I2C/(2*Tmaster)?

The I2C interface doc says:

● Standard mode:
thigh = CCR * tCK <- we should count by this
tlow = CCR * tCK
● Fast mode:
If DUTY = 0:
thigh = CCR * tCK
tlow = 2 * CCR * tCK <- formula in the article
If DUTY = 1: (to reach 400 kHz)
thigh = 9 * CCR * tCK
tlow = 16 * CCR * tCK

Please explain, thank you.

Login or register to post comments

STM8 i2c master

PUBLISHED WED, 06/01/2016 - 09:27 BY SELEVO@MAIL.RU

Thanks
, it would be nice to see the pitfalls that needed to be eliminated for normal operation. I'm really looking forward to the article about I2C slave on STM8,
for example, emulation of MCP23017 or PCF8574
. I can throw 500 rubles into the piggy bank.
By the way, the pictures in the article are not visible.

Login or register to post comments

STM8S_StdPeriph_Driver

PUBLISHED TUE, 09/26/2017 - 18:06 BY NIK S

Hello, I'm trying to use **STM8S_StdPeriph_Driver** but I can't read the data right away.

> *To be honest, I had to tinker with the I2C module for a long time - it just wouldn't work the way it should until I read the errata and fixed some other small but insidious errors that migrated into my program from the example provided on the official STMicroelectronics website.*

I would like to know what errors you encountered in the official examples?

Login or register to post comments


80