Главная

Сайт:
STM8

Сегодня я хотел бы кратко рассказать о модуле I2C на микроконтроллере STM8S003F и поделиться своими наработками в этой области.

Признаться честно, долго пришлось повозиться с модулем I2C- он ни в какую не хотел работать так, как нужно, пока я не прочитал errata и не устранил еще некоторые мелкие, но коварные ошибки, которые перекочевали в мою программу из примера, предоставленного на официальном сайте STMicroelectronics.

В этой статье я покажу как настроить I2C в режиме мастера без прерываний. В самом конце статьи можно будет скачать готовый аппартный драйвер I2C для STM8. В следующей статье попробуем подключить часы реального времени DS1307 и выводить время через UART. Возможно, в дальнейшем будет пример на прерываниях, или может кто-нибудь захочет поделиться своими наработками – милости прошу.

Для тех, кто желает разобраться в принципе работы I2C интерфейса, советую почитать краткую и в тоже время, полезную статью: **Интерфейсная шина IIC (I2C)**

Также советую почитать полезное замечание: **Как я побеждал I2C (STM8L)**

Еще нужно будет почитать статью по **настройке модуля UART** и ознакомиться с даташитами **STM8S003** и **Reference manual**.

Рассмотрим основные возможности модуля I2C:

- Поддержка режима Master и Slave(ведущий и ведомый)
- Генерация и определение 7- и 10-битного адресов
- Поддержка разных скоростей передачи:
  - Стандартный режим 100 кГц
  - Высокоскоростной режим 400 кГц

Как обычно начинаемс инициализации.

1. Задаем частоту тактирования модуля I2C. Это частота Fmaster, которая снимается перед делителем CPUDIV. Если Вы читали прошлую статью по настройке модуля UART, то знаете, что по умолчанию в качестве источника тактирования выбран RC-генератор (16МГц), с делением частоты на 8. Т.е. по умолчанию Fmaster= 2МГц. Значение Fmaster записывается в регистр **I2C_FREQR** в МГц. Значения, которые могут быть записаны в данный регистр, находятся в диапазоне 1…24 МГц.

2. Дальше нужно решить в каком режиме мы будем работать. DS1307 работает только в стандартном режиме (100 кГц), поэтому его и выбираем. Стандартный режим выбирается сбросом бита **F/S** в регистре **I2C_CCRH.**

3. Настраиваем регистры управления тактированием **I2C_CCRL** и **I2C_CCRH**. Эти регистры определяют длительность импульса тактировании SCL и длительность паузы.

I2C_CCRL

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

I2C_CCRH

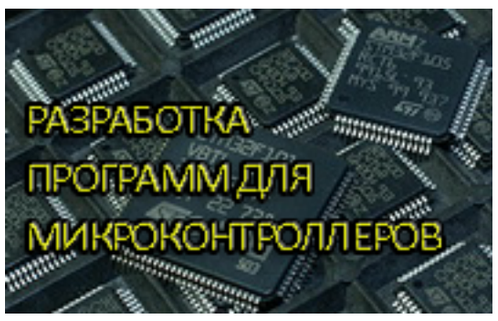
7	6	5	4	3	2	1	0
F/S	DUTY	Reserved		CCR[11:8]			
rw	rw			rw			

Коэффициент **CCR** вычисляется по разным формулам (они описаны в reference manual), в зависимости от выбранного режима работы. Для стандартного режима он вычисляется по формуле:

CCR= Period_I2C/(2*Tmaster), где

Period_I2C- период импульсов SCLна I2C шине, в стандартном режиме минимальный период

Микроконтроллеры
Схемы
Форум
Начинающим
Справочник
Программы
Расчеты
Технологии
Книги



Установка и запуск веб-сервера Apache на Orange Pi Zero
Настройка сети на Orange Pi Zero
Знакомство с одноплатным миникомпьютером Orange Pi Zero
Настройка модуля CAN на микроконтроллере STM32F103. Часть 1
Урок 5. Первая программа на PIC12F675
Драйвер для часов реального времени DS1307 на STM8
Настройка I2C на микроконтроллере STM8
Настройка UART на

равен 1/100кГц,

Tmaster– период частоты тактирования периферии, Tmaster= 1/Fmaster= 1/2 000 000 Гц.

Таким образом формула принимает вид:

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

Fmaster - частота периферии в МГц,

Fi2c– частота I2Cшины.

Бит **DUTY** позволяет выбрать коэффициент заполнения для высокоскоростного режима.

4. Запрограммировать максимальное время нарастания сигнала SCL. По спецификации время нарастания сигнала SCL в стандартном режиме не должно превышать 1000 нс. Время нарастания задается в регистре **TRISE**.

Значение регистра вычисляется по формуле:

TRISE= (1000 нс/Tmaster) + 1.

Если частота Fmaster=2МГц, то период равен 500нс, значит значение регистра:

TRISE = (1000 нс/500нс) + 1 = 3

5. Включаем модуль I2C установкой бита **PE**=1 в регистре **I2C_CR1.**

6. Устанавливаем бит, разрешающий подтверждение принятого байта **ACK**=1 в регистре **I2C_CR2**

Опишу кратко *регистры статуса и управления*, с которыми нам придется дальше работать.

I2C_CR2

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

- **SWRST** – программный сброс модуля I2C
- **POS**– этот бит устанавливается только в случае, когда нужно принять два байта (N=2). Он должен быть установлен до начала приема данных. В этом случае импульс NACKсформируется после приема второго байта, и ACK можно сразу же сбросить сразу после отправки адреса.
- **ACK** – разрешает отсылать подтверждения от мастера после приема каждого байта
- **STOP** – дает указание мастеру сгенерировать стоповую посылку после приема текущего байта
- **START**– дает указание мастеру сгенерировать стартовую посылку

I2C_SR1

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

- **TXE** – регистр данных DR пуст
- **RXNE** – в регистре данных DRсодержится принятый байт
- **STOPF**– обнаружена стоп-посылка на шине
- **ADD10** – мастер отправил первый байт 10-битного адреса
- **BTF**– устанавливается когда в приемный регистр DR поступил байт, но не был прочитан, и уже пришел новый байт в сдвиговый регистр
- **ADDR** – мастер передал адрес
- **SB**– мастер сгенерировал старт-посылку

I2C_SR2

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** – выход из спящего режима
- **OVR** – ошибка переполнения регистра данных
- **AF**– ошибка подтверждения отправленного мастером байта
- **ARLO** – потеря арбитража (на шине, где более одного мастера)
- **BERR**– ошибка шины (например когда послан неуместный старт или стоп)

I2C_SR3

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

- TRA– если данные приняты = 0, если переданы = 1
- BUSY – сигнализирует о занятости шины
- MSL – показывает в каком режиме находится модуль I2C. 1 – мастер, 0 – слейв

Также есть регистр настройки прерываний I2C_ITR, который мы не будем настраивать в данном примере. Также для слейва имеются регистры, в которые записывается собственный адрес I2C устройства - **I2C_OARL, I2C_OARH.**

Дальше я немного перевел для вас информации из reference manual. Если где-то ошибся, то прошу меня поправить.

Режим мастера

В режиме мастера I2Cинтерфейс инициирует передачу данных и генерирует clockсигнал

тактирования. Каждая передача начинается СТАРТ условием и оканчивается СТОП условием. Режим мастера включается сразу же после генерации СТАРТ.

Входная частота должна быть не меньше:

- 1 MHz в стандартном режиме
- 4 MHz в быстром режиме

Стартовое условие (стартовая посылка)

Установка бита STARTприводит к генерации стартовой посылки и переключает в режим мастера, при условии, что BUSY= 0.

Заметка: В режиме мастера установка START бита генерирует посылку рестарта (повторного старта) в конце текущего передаваемого байта.

Сразу же после стартовой посылки устанавливает бит SB и генерируется прерывание, если оно разрешено ITEVTEN= 1.

Далее мастер ожидает чтение регистра SR1, после чего в регистр DR записывается адрес слейв-устройтва (slaveaddress).

Передача слейв-адреса

Слейв-адрес передается в SDA линию из сдвигового регистра.

● В 10-битном режиме адресации отправка сопровождается следующими событиями:

Устанавливается бит ADD10 аппаратно и генерируется прерывание, если оно разрешено в ITEVTEN.

Затем мастер ожидает чтения регистра SR1, после чего в DR записывается вторая часть адреса.

Устанавливается бит ADDRаппаратно и генерируется прерывание, если оно разрешено в ITEVTEN.

Затем мастер ожидает чтения регистра SR1 и регистра SR3, что позволяет сбросить бит ADDR и продолжить передачу.

● В 7-битном режиме адресации отправляется только один байт.

Сразу же после отправки байта адреса устанавливается аппаратно бит ADDR и генерируется прерывание, если оно разрешено в ITEVTEN.

Затем мастер ожидает чтения регистра SR1 и регистра SR3.

Мастер может решать перейти в режим передачи или приема в соответствии с 0-м битом слейв-адреса.

Если нулевой бит = 0, то передача, если = 1, то прием.

● В режиме 10-битной адресации:

– Для входа в режим передачи мастер отправляет заголовок (11110xx0) и слейв-адрес (где xx– это два старших бита адреса).

– Для входа в режим приема мастер отправляет заголовок (11110xx0) слейв-адрес. Затем оправляется повторная реСТАРТ-посылка,затем заголовок (11110xx1).

Бит *TRA* показывает в каком режиме находится мастер: прием или передача.

Мастер в режиме передатчика

После передачи адреса и очистки бита ADDR, мастер отправляет байт из регистра DR на SDA линию через внутренний сдвиговый регистр.

Мастер ожидает до тех пор, пока в DR запишется первый байт данных (событие EV8_1).

Когда принят сигнал (импульс) подтверждения:

● Устанавливается аппаратно бит TXE и генерируется прерывание, если оно разрешено в ITEVTEN, а также устанавливается бит ITBUFEN.

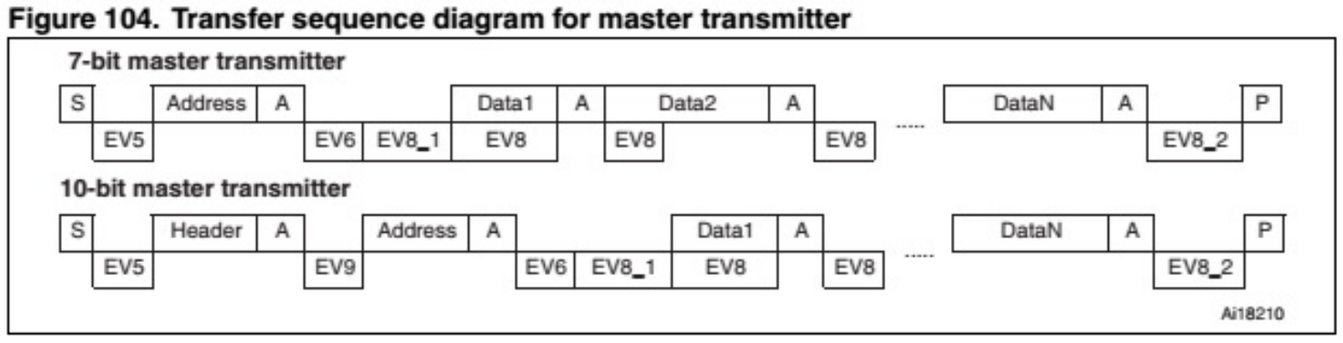
● Если TXE установлен и байт данных не был записан в DR до окончания следующей передачи данных, устанавливается бит BTF, и интерфейс ожидает, пока он не будет очищен, что делается чтением регистра SR1 и записью в DR, SCL удерживается в низком состоянии.

Завершения связи (передачи)

После записи последнего байта в DR, устанавливают бит STOP, который вызывает генерацию СТОП посылки (событие EV8_2). Интерфейс автоматически переходит в режим слейв. (MSLбит сбрасывается).

Заметка: Стоп-посылка должна быть запрограммирована во время события EV8_2, когда TXE или BTF установлены.

Figure 104. Transfer sequence diagram for master transmitter



S= Старт,

Sr= Повторный старт,

P= Стоп,

A= Подтвеждение,

NA= неподтверждение,

EVx= событие

EV5:SB=1 очищается чтением регистра SR1, далее записывается адрес в DR.

EV6:ADDR=1, очищается чтением регистра SR1 и последующим чтением SR3.

EV8_1:TXE=1, сдвиговый регистр пуст, регистр данных пуст, запись данных в DR.

EV8:TXE=1, сдвиговый регистр не пуст, регистр данных DR пуст, очищается записью в DR.

EV8_2:TXE=1, BTF= 1, Программируется STOP запрос. TXEи BTF очищаются аппаратно после генерации стоп-посылки.

EV9:ADD10=1, очищается чтением регистра SR1, далее записывается регистр DR.

Событие **EV8** должно быть выполнено до конца передачи текущего байта. В противном случае рекомендуется использовать BTF вместо TXE с замедлением связи.

Мастер в режиме приемника

После передачи адреса и очистки бита ADDR I2C интерфейс входит в режим приемника. В этом режиме интерфейс принимает байты из SDA линии в DR регистр через внутренний сдвиговый регистр.

После каждого байта интерфейс генерирует последовательность:

- Подтверждающий импульс, если бит ACK установлен
- Установка бита RXNE и генерация прерывания, если биты ITEVTEN и ITBUFEN установлены.

Если бит RXNE установлен и данные не были прочитаны из DR до того, как был принят следующий байт, аппаратно устанавливается бит BTF и интерфейс ожидает, пока этот бит будет очищен чтением I2C_SR1 и I2C_DR, SCL удерживается в низком состоянии.

Завершение соединения

Метод 1: *Этот метод подходит в том случае, если I2C использует прерывания, которые имеют наивысший приоритет в приложении.*

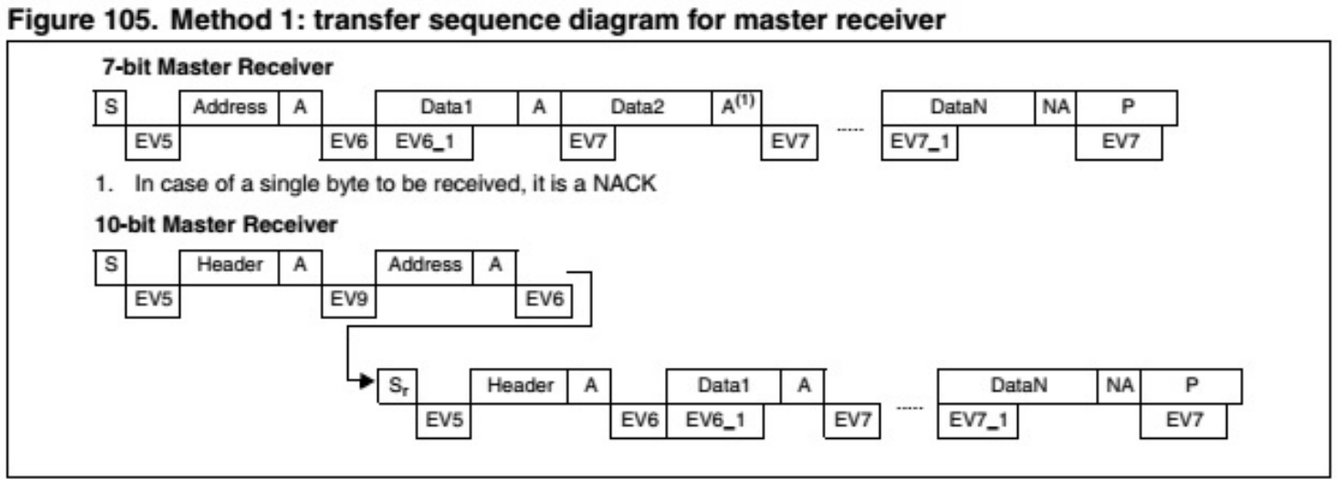
Мастер отправляет NACK в конце последнего байта от слейва.

После приема этого NACK слейв освобождает линии SCL и SDA. Затем мастер может отправить Стоп или Рестарт посылку.

- В случает генерации NACK импульса после последнего принятого байта, бит ACK должен быть очищен точно после чтения предпоследнего байта (после предпоследнего события RXNE).
- В случае генерации Стоп или Рестарта приложение должно установить бит STOP/START сразу после чтения предпоследнего байта (после предпоследнего события RXNE).
- В случае приема одного байта, подтверждение деактивируется и генерируется СТОП после события EV6 (в EV6 сразу после очистки ADDR).

После генерации СТОП посылки интерфейс автоматически переходит в режим слейва. (MSL= 0).

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



EV5:*SB=1, очищается чтением SR1 с последующей записью в DR.*

EV6:*ADDR=1, очищается чтением SR1 и последующим чтением SR3. В 10-битном режиме мастера-приемника эта последовательность должна следовать после записи в CR2 бита START= 1.*

EV6_1:*нет флагов данного события, используется только для однобайтного приема. Программируется ACK=0 и STOP=1 после очистки ADDR.*

EV7:*RXNE=1, очищается чтением DR.*

EV7_1:*RXNE=1, очищается чтением DR, программируется ACK=0 и STOP запрос*

EV9:*ADD10=1, очищается чтением SR1 с последующей записью в DR.*

1.Если регистр DR заполнен, прием следующих данных выполняется после очистки события ***EV7***.

2. EV5, EV6 и EV9 события удерживают SCL в низкоуровневом состоянии до конца соответствующей последовательности действий в приложении.

3. EV7 программная последовательность должна быть завершена до конца передачи текущего байта. В противном случае рекомендуется использовать BTF вместо RXNE, тем самым замедляя скорость связи.

4. Последовательность EV6_1 или EV7_1 должна быть завершена до импульса подтверждения ACK текущего байта.

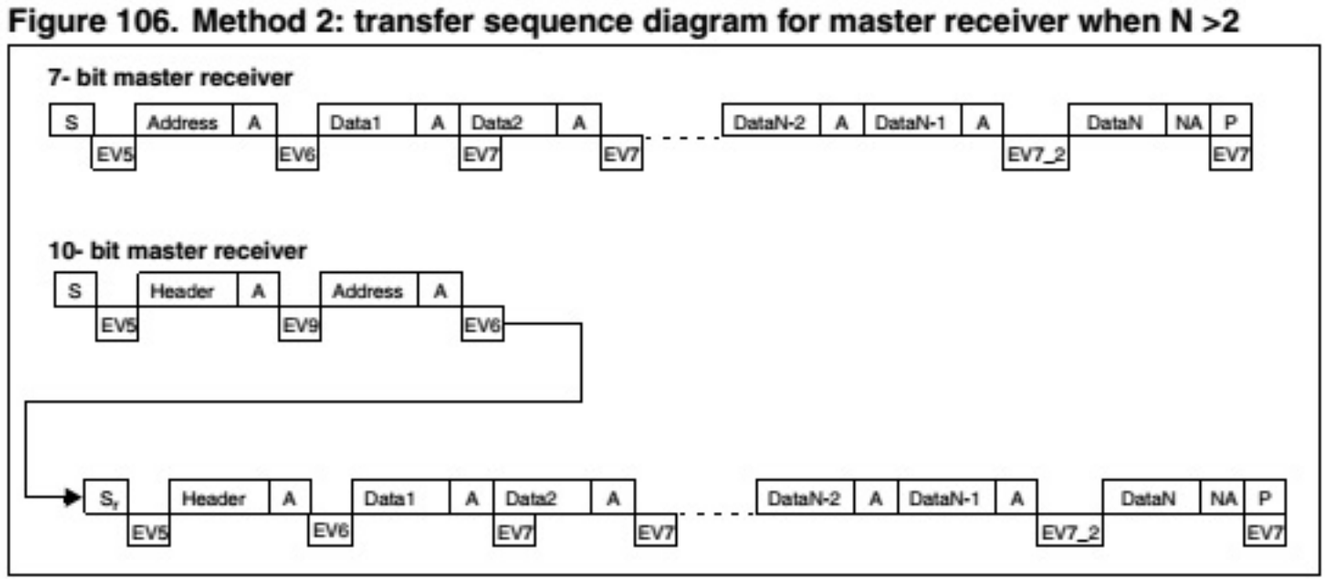
Метод 2: *Этот метод подходит для тех случаев, когда используются прерывания, которые не имеют наивысшего приоритета в приложении и когда I2C используется в режиме опроса.*

● DataN_2 не читается, так что после DataN_1 соединение замедляется (оба бита установлены RxNE и BTF).

● Далее, бит ACK должен быть очищен до чтения DataN-2 из DR для уверенности, что этот бит был очищен до импульса подтверждения DataN.

● После этого сразу после чтения DataN_2, приложения должно установить бит STOP/START и прочитать DataN_1. После установки RXNE читается DataN.

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



EV5: SB=1, *очищается чтением* SR1 *с последующей записью в* DR.

EV6:ADDR1, *очищается чтением* SR1 *с последующим чтением* SR3.

В 10-битном режиме мастера-приемника эта последовательность должна следовать после записи в CR2 бита START= 1.

EV7: RxNE=1, *очищается чтением* DR.

EV7_2: BTF= 1, DataN-2 в DR, а DataN-1 в сдвиговом регистре, программируется ACK= 0, читаются данные DataN-2 из DR. Устанавливается STOP= 1, читается DataN-1.

EV9: ADD10= 1, *очищается чтением* SR1 *с последующей записью в* DR.

Когда остается прочитать 3 байта (N>2):

● RxNE= 1 => ничего не делать (DataN-2 не читаем).

● DataN-1 принят

● BTF= 1 потому что сдвиговый и регистр данных заполнены: DataN-2 в DR и DataN-1 в сдвиговом регистре => SCL удерживается в низком состоянии: никакие другие данные не будут получены из шины.

● Сбрасываем ACK= 0

● Читаем DataN-2 из DR=> Это позволяет DataN попасть в сдвиговый регистр

● DataN принят (с посылкой NACK)

● Программируется бит START/STOP

● Читаются данные DataN-1

● Ожидается установка RxNE= 1

● Читаются данные DataN

Процедура, описанная выше, подходит для случая, когда нужно принять байт N>2.

Когда нужно принять один (N=1) байт или два(N=2), обработка будет отличаться:

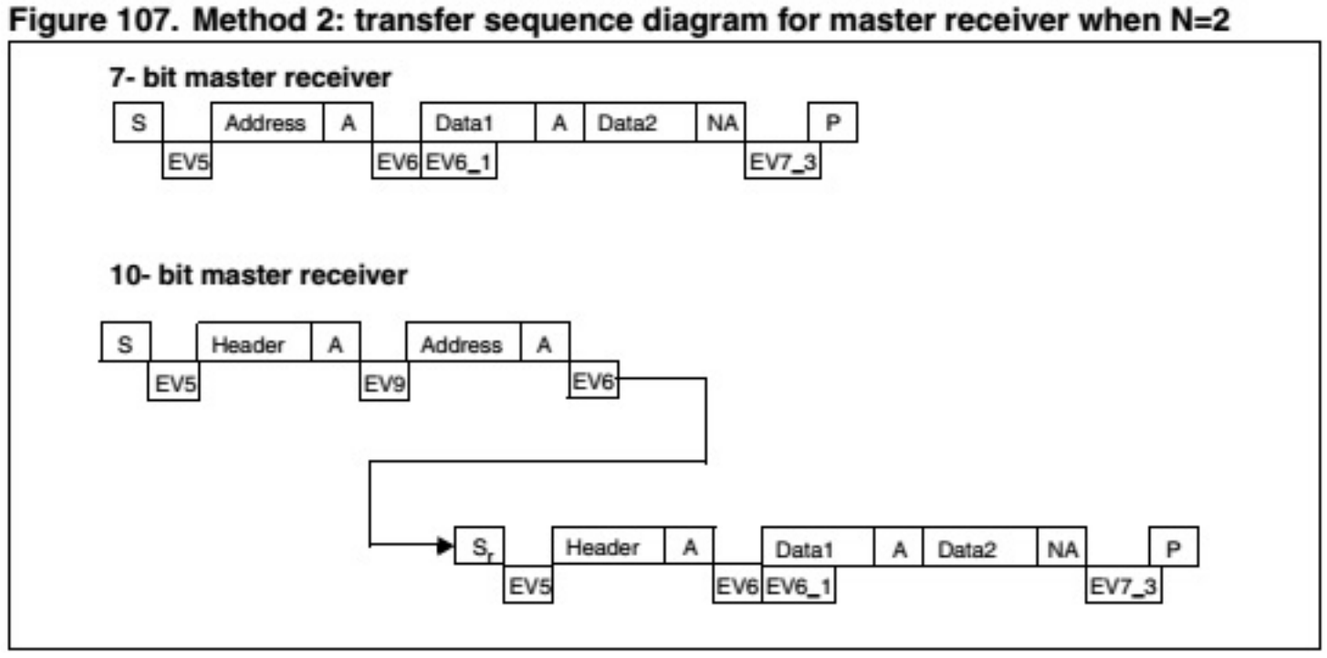
● ***Случай N=1***

- Ожидаем установки ADDR, сбрасывает бит ACK=0.
- Сбрасывает ADDR=0
- Устанавлием бит STOP/START.
- Читаем данные, после того, как установится RxNE.

● ***Случай N=2***

- Устанавливаются POS=1 и ACK=1
- Ожидается установка ADDR=1
- Очищается ADDR
-Очищается ACK
- Ожидается установка BTF=1
-Программируется STOP=1
- Читаем DR дважды

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



EV5: SB=1, *очищается чтением* SR1 *с последующей записью в* DR.

EV6:ADDR1, *очищается чтением* SR1 *с последующим чтением* SR3.

В 10-битном режиме мастера-приемника эта последовательность должна следовать после записи в CR2 бита START= 1.

EV6_1:*нет флагов данного события. Подтверждение должно быть отключено сразу после события EV6, после чего ADDRочищается*

EV7_3: BTF= 1, программируется STOP=1, дважды читается регистр DR сразу после установки бита STOP.

EV9: ADD10= 1, *очищается чтением* SR1 *с последующей записью в* DR.

EV5, EV6 и EV9 события удерживают SCLв низкоуровневом состоянии до конца соответствующей последовательности действий в приложении.

EV6_1 *последовательность должна быть завершена до установки импульса подтверждения ACK для текущего передаваемого байта.*

Ну и наконец **пример кода**:

```
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
```

```
  I2C_CR2_STOP = 1;
  //Ждем выполнения условия СТОП
  wait_event(I2C_CR2_STOP, 1);

  return I2C_SUCCESS;
}

//*************************************************************************
// Чтение регистра slave-устройства
// Start -> Slave Addr -> Reg. addr -> Restart -> Slave Addr <- data ... -> Stop
//*************************************************************************
t_i2c_status i2c_rd_reg(unsigned char address, unsigned char reg_addr,
                        char * data, unsigned char length)
{

  //Ждем освобождения шины I2C
  wait_event(I2C_SR3_BUSY, 10);

  //Разрешаем подтверждение в конце посылки
  I2C_CR2_ACK = 1;

  //Генерация СТАРТ-посылки
  I2C_CR2_START = 1;
  //Ждем установки бита SB
  wait_event(!I2C_SR1_SB, 1);

  //Записываем в регистр данных адрес ведомого устройства
  I2C_DR = address & 0xFE;
  //Ждем подтверждения передачи адреса
  wait_event(!I2C_SR1_ADDR, 1);
  //Очистка бита ADDR чтением регистра SR3
  I2C_SR3;

  //Ждем освобождения регистра данных RD
  wait_event(!I2C_SR1_TXE, 1);

  //Передаем адрес регистра slave-устройства, который хотим прочитать
  I2C_DR = reg_addr;
  //Ловим момент, когда DR освободился и данные попали в сдвиговый регистр
  wait_event(!(I2C_SR1_TXE && I2C_SR1_BTF), 1);

  //Генерация СТАРТ-посылки (рестарт)
  I2C_CR2_START = 1;
  //Ждем установки бита SB
  wait_event(!I2C_SR1_SB, 1);

  //Записываем в регистр данных адрес ведомого устройства и переходим
  //в режим чтения (установкой младшего бита в 1)
  I2C_DR = address | 0x01;

  //Дальше алгоритм зависит от количества принимаемых байт
  //N=1
  if(length == 1){
    //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;
```

```
246.     //Читаем N-1 байт
247.     *data++ = I2C_DR;
248.     //Заплатка из Errata
249.     __enable_interrupt();
250.     //Ждем, когда N-й байт попадет в DR из сдвигового регистра
251.     wait_event(!I2C_SR1_RXNE, 1);
252.     //Читаем N байт
253.     *data++ = I2C_DR;
254.   }
255.
256.   //Ждем отправки СТОП посылки
257.   wait_event(I2C_CR2_STOP, 1);
258.   //Сбрасывает бит POS, если вдруг он был установлен
259.   I2C_CR2_POS = 0;
260.
261.   return I2C_SUCCESS;
262. }
```

Драйвер I2C для STM8 в IAR

ОПУБЛИКОВАНО ВС, 12/06/2015 - 19:41 ПОЛЬЗОВАТЕЛЕМ ALEX2015

Добрый день.
Почему расчет CCR для стандартного режима происходит по формуле
CCR= Period_I2C/(2*Tmaster)?

В доке на интерфейс I2C написано:

● Standard mode:
thigh = CCR * tCK <- мы должны считать по этой
tlow = CCR * tCK
● Fast mode:
If DUTY = 0:
thigh = CCR * tCK
tlow = 2 * CCR * tCK <-формула в статье
If DUTY = 1: (to reach 400 kHz)
thigh = 9 * CCR * tCK
tlow = 16 * CCR * tCK

Объясните пожалуйста, Спасибо.

ОПУБЛИКОВАНО СР, 06/01/2016 - 09:27 ПОЛЬЗОВАТЕЛЕМ SELEVO@MAIL.RU

Спасибо
ещё бы было указано про подводные камни которые надо было устранить для нормальной работу.
Очень жду статьи про I2C slave на STM8
например эмуляцию MCP23017 или PCF8574
Могу закинуть в копилку 500р.
Кстати, не видны картинки в статье.

ОПУБЛИКОВАНО ВТ, 09/26/2017 - 18:06 ПОЛЬЗОВАТЕЛЕМ NIK S

Привет, я пробую использовать **STM8S_StdPeriph_Driver** но сходу считать данные не получилось.

> *Признаться честно, долго пришлось повозиться с модулем I2C- он ни в какую не хотел работать так, как нужно, пока я не прочитал errata и не устранил еще некоторые мелкие, но коварные ошибки, которые перекочевали в мою программу из примера, предоставленного на официальном сайте STMicroelectronics.*

Хотелось бы узнать с какими ошибками пришлось столкнуться в официальных ппримерах?