

This website uses cookies to improve your experience and to gather page view statistics. This site does not collect user information. [Accept & Close](#) [Read More](#)

Using hardware and software to make new stuff

Search

Categories

- [3D Printing](#) (1)
- [Affinity Photo](#) (1)
- [Aide-memoir](#) (1)
- [Analog](#) (1)
- [Ansible](#) (3)
- [Dates to Remember](#) (3)
- [Docker](#) (1)
- [Electronics](#) (144)
- [ESP8266](#) (12)
- [FPGA](#) (4)
- [Garden](#) (1)
- [General](#) (8)
- [Home Built CPU](#) (6)
- [Informatica](#) (1)
- [Internet of Things](#) (8)
- [iOS](#) (2)
- [KiCad](#) (4)
- [MSP430](#) (2)
- [Netduino](#) (49)
- [NuttX](#) (9)
- [Photography](#) (3)
- [Pico](#) (10)
- [Raspberry Pi](#) (17)
- [Silverlight](#) (6)
- [Software Development](#) (88)
- [STM32](#) (5)
- [STM8](#) (54)
- [Tips](#) (5)

Archives

- [July 2024](#)
- [June 2024](#)
- [May 2024](#)
- [April 2024](#)
- [March 2024](#)
- [February 2024](#)
- [January 2024](#)
- [December 2023](#)
- [November 2023](#)
- [October 2023](#)

This website uses cookies to improve your experience and to gather page view statistics. This site does not collect user information. [Accept & Close](#) [Read More](#)

- ◊ [April 2020](#)
- ◊ [February 2020](#)
- ◊ [January 2020](#)
- ◊ [July 2019](#)
- ◊ [February 2018](#)
- ◊ [July 2017](#)
- ◊ [June 2017](#)
- ◊ [April 2017](#)
- ◊ [March 2017](#)
- ◊ [February 2017](#)
- ◊ [October 2016](#)
- ◊ [September 2016](#)
- ◊ [July 2016](#)
- ◊ [June 2016](#)
- ◊ [May 2016](#)
- ◊ [April 2016](#)
- ◊ [March 2016](#)
- ◊ [January 2016](#)
- ◊ [December 2015](#)
- ◊ [November 2015](#)
- ◊ [October 2015](#)
- ◊ [August 2015](#)
- ◊ [June 2015](#)
- ◊ [May 2015](#)
- ◊ [April 2015](#)
- ◊ [March 2015](#)
- ◊ [February 2015](#)
- ◊ [January 2015](#)
- ◊ [December 2014](#)
- ◊ [October 2014](#)
- ◊ [September 2014](#)
- ◊ [August 2014](#)
- ◊ [July 2014](#)
- ◊ [June 2014](#)
- ◊ [May 2014](#)
- ◊ [March 2014](#)
- ◊ [February 2014](#)
- ◊ [January 2014](#)
- ◊ [December 2013](#)
- ◊ [November 2013](#)
- ◊ [October 2013](#)
- ◊ [September 2013](#)
- ◊ [August 2013](#)
- ◊ [July 2013](#)
- ◊ [June 2013](#)
- ◊ [May 2013](#)
- ◊ [April 2013](#)
- ◊ [March 2013](#)
- ◊ [January 2013](#)
- ◊ [November 2012](#)
- ◊ [October 2012](#)
- ◊ [September 2012](#)
- ◊ [August 2012](#)
- ◊ [July 2012](#)
- ◊ [June 2012](#)
- ◊ [May 2012](#)

This website uses cookies to improve your experience and to gather page view statistics. This site does not collect user information. [Accept & Close](#) [Read More](#)

- [October 2011](#)
- [September 2011](#)
- [July 2011](#)
- [June 2011](#)
- [May 2011](#)
- [April 2011](#)
- [March 2011](#)
- [February 2011](#)
- [January 2011](#)
- [December 2010](#)
- [April 2010](#)

STM8S I2C Master Devices

I have finally managed to dig out the TMP102 temperature sensor from the back of the electronic breakout board cupboard. Why am I interested in this sensor, well it is about the only I2C device I actually own and I2C is one of the few areas I have not really looked at on the STM8S.

This article will explore the basics of creating a I2C master device using the STM8S as the bus master and the TMP102 as the slave device.

I2C Protocol

[I2C \(Inter-Integrated Circuit\)](#) is a communication protocol allowing bi-directional communication between two or more devices over two wires. The protocol allows a device to be in one of four modes:

1. Master transmit
2. Master receive
3. Slave transmit
4. Slave receive

The Wikipedia article contains a good description of the protocol and the various modes and the bus characteristics.

For the purposes of this post the STM8S will need to be in master mode as it will be controlling the communication flow with the temperature sensor which is essentially a dumb device.

Communicating with the TMP102

The TMP102 is a simple device which returns two bytes of data which represent the current reading from a temperature sensor. The process for reading the temperature is as follows:

1. Master device transmits the slave address on the SDA line along with a bit indicating that it wishes to read data from the slave device
2. Master device enters *Master Receive* mode
3. Slave device transmits two bytes of data representing the temperature
4. Master device closes down the communication sequence

The temperature can be calculated according to the following formula:

Temperature in centigrade = (((byte₀ * 256) + byte₁) / 16) * 0.0625

Wiring up the TMP102

This website uses cookies to improve your experience and to gather page view statistics. This site does not collect user information. [Accept & Close](#) [Read More](#)

SCK	Microcontroller SCK
Vcc	3.3 V
GND	Ground
ADR0	Ground

This should result in the breakout having an I2C address of 0x48.

Netduino Code

The Netduino 3 runs the .NET Microframework (NETMF). This has built in class for communicating over I2C. A simple application will give a reference point for how the protocol should work.

This website uses cookies to improve your experience and to gather page view statistics. This site does not collect user information. [Accept & Close](#) [Read More](#)

```

5 namespace TMP102
6 {
7     public class Program
8     {
9         public static void Main()
10        {
11            //
12            // Create a new I2C device for the TMP102 on address 0x48 with t
13            // running at 50 KHz.
14            //
15            I2CDevice tmp102 = new I2CDevice(new I2CDevice.Configuration(0x48
16            //
17            // Create a transaction to read two bytes of data from the TMP10
18            //
19            byte[] buffer = new byte[2];
20            I2CDevice.I2CTransaction[] reading = new I2CDevice.I2CTransaction
21            reading[0] = I2CDevice.CreateReadTransaction(buffer);
22            while (true)
23            {
24                //
25                // Read the temperature.
26                //
27                int bytesRead = tmp102.Execute(reading, 100);
28                //
29                // Convert the reading into Centigrade and Fahrenheit.
30                //
31                int sensorReading = ((buffer[0] << 8) | buffer[1]) >> 4;
32                double centigrade = sensorReading * 0.0625;
33                double fahrenheit = centigrade * 1.8 + 32;
34                //
35                // Display the readings in the debug window and pause before
36                //
37                Debug.Print(centigrade.ToString() + " C / " + fahrenheit.ToSt
38                Thread.Sleep(1000);
39            }
40        }
41    }
42 }

```

Running this on the Netduino gives the following output in the debug window:

```

20.6875 C / 69.23750000000011 F
20.625 C / 69.125 F
20.6875 C / 69.23750000000011 F
20.6875 C / 69.23750000000011 F
20.625 C / 69.125 F
20.6875 C / 69.23750000000011 F
20.625 C / 69.125 F
20.625 C / 69.125 F
20.625 C / 69.125 F
20.625 C / 69.125 F
20.625 C / 69.125 F

```

Hooking up the Saleae Logic 16 gives the following output for a single reading:

This website uses cookies to improve your experience and to gather page view statistics. This site does not collect user information. [Accept & Close](#) [Read More](#)

I2C communication with a TMP102 and a Netduino 3

STM8S Implementation

The NETMF class used above hides a lot of the low level work which the STM8S will have to manage. In order to communicate with the TMP102 the STM8S will have to perform the following:

1. Enter master transmit mode
2. Send the 7-bit address of the sensor
3. Send a single bit indicating that we want to read from the sensor
4. Wait for the slave to respond with an ACK
5. Enter master receiver mode
6. Read the bytes from the slave device. Send an ACK signal for all bytes except the last one.
7. Send a NAK signal at the end of the sequence

The first task is to initialise the I2C system on the STM8S:

```

1  //
2  //  Initialise the I2C system.
3  //
4  void InitialiseI2C()
5  {
6      I2C_CR1_PE = 0;                // Diabie I2C before configuration s
7      //
8      //  Setup the clock information.
9      //
10     I2C_FREQR = 16;                // Set the internal clock frequency
11     I2C_CCRH_F_S = 0;              // I2C running is standard mode.
12     I2C_CCRL = 0xa0;               // SCL clock speed is 50 KHz.
13     I2C_CCRH_CCR = 0x00;
14     //
15     //  Set the address of this device.
16     //
17     I2C_OARH_ADDMODE = 0;          // 7 bit address mode.
18     I2C_OARH_ADDCONF = 1;         // Docs say this must always be 1.
19     //
20     //  Setup the bus characteristics.
21     //
22     I2C_TRISER = 17;
23     //
24     //  Turn on the interrupts.
25     //
26     I2C_ITR_ITBUFEN = 1;           // Buffer interrupt enabled.
27     I2C_ITR_ITEVTEN = 1;          // Event interrupt enabled.
28     I2C_ITR_ITERREN = 1;
29     //
30     //  Configuration complete so turn the peripheral on.
31     //
32     I2C_CR1_PE = 1;
33     //
34     //  Enter master mode.
35     //

```

This website uses cookies to improve your experience and to gather page view statistics. This site does not collect user information. [Accept & Close](#) [Read More](#)

Some of the initialisation for the I2C bus needs to be performed whilst the peripheral is disabled, notably the setup of the clock speed. The method above disables the I2C bus, sets up the clock and addressing mode, turns on the interrupts for the peripheral and then enables the I2C bus. Finally, the method sets up the system to transmit ACKs following data reception and then sends the Start bit.

The communication with the slave device is handled by an Interrupt Service Routine (ISR). The initialisation method above will have taken control of the bus and set the start condition. An interrupt will be generated once the start condition has been set.

The master then needs to send the 7-bit address followed by a 1 to indicate the intention to read data from the bus. These two are normally combined into a single byte, the top 7-bits containing the device address and the lower bit indicating the mode (read – 1 or write – 0).

```
1  if (I2C_SR1_SB)
2  {
3      //
4      //  Master mode, send the address of the peripheral we
5      //  are talking to.  Reading SR1 clears the start condition.
6      //
7      reg = I2C_SR1;
8      //
9      //  Send the slave address and the read bit.
10     //
11     I2C_DR = (DEVICE_ADDRESS << 1) | I2C_READ;
12     //
13     //  Clear the address registers.
14     //
15     I2C_OARL_ADD = 0;
16     I2C_OARH_ADD = 0;
17     return;
18 }
```

This above code checks the status registers to see if the interrupt has been generated because of a start condition. If it has then the STM8S is setup to send the address of the TMP102 along with the read bit

Assuming that no error conditions are generated, the next interrupt generated will indicate that the address has been sent to the slave successfully. This condition is dealt with by reading two of the status registers and clearing the address registers:

```
1  if (I2C_SR1_ADDR)
2  {
3      //
4      //  In master mode, the address has been sent to the slave.
5      //  Clear the status registers and wait for some data from the slave.
6      //
7      reg = I2C_SR1;
8      reg = I2C_SR3;
9      return;
10 }
```

At this point the address has been sent successfully and the I2C peripheral should be ready to start to receive data.

This website uses cookies to improve your experience and to gather page view statistics. This site does not collect user information. [Accept & Close](#) [Read More](#)

The master device can then continue to hold control of the bus or it can send a STOP signal indicating that the flow of communication has ended.

These two conditions are dealt with by resetting the *I2C_CR2_ACK* bit and setting the *I2C_CR2_STOP* bit.

The full ISR for the I2C bus looks like this:

```

1  //
2  //  I2C interrupts all share the same handler.
3  //
4  #pragma vector = I2C_RXNE_vector
5  __interrupt void I2C_IRQHandler()
6  {
7      if (I2C_SR1_SB)
8      {
9          //
10         //  Master mode, send the address of the peripheral we
11         //  are talking to. Reading SR1 clears the start condition.
12         //
13         reg = I2C_SR1;
14         //
15         //  Send the slave address and the read bit.
16         //
17         I2C_DR = (DEVICE_ADDRESS << 1) | I2C_READ;
18         //
19         //  Clear the address registers.
20         //
21         I2C_OARL_ADD = 0;
22         I2C_OARH_ADD = 0;
23         return;
24     }
25     if (I2C_SR1_ADDR)
26     {
27         //
28         //  In master mode, the address has been sent to the slave.
29         //  Clear the status registers and wait for some data from the slave.
30         //
31         reg = I2C_SR1;
32         reg = I2C_SR3;
33         return;
34     }
35     if (I2C_SR1_RXNE)
36     {
37         //
38         //  The TMP102 temperature sensor returns two bytes of data
39         //
40         _buffer[_nextByte++] = I2C_DR;
41         if (_nextByte == 1)
42         {
43             I2C_CR2_ACK = 0;
44             I2C_CR2_STOP = 1;
45         }
46         return;
47     }
48 }
```


This website uses cookies to improve your experience and to gather page view statistics. This site does not collect user information. [Accept & Close](#) [Read More](#)

ready through this port. Add to this some initialisation code and the full application looks as follows:

```
1  //
2  // This application demonstrates the principles behind developing an
3  // I2C master device on the STM8S microcontroller. The application
4  // will read the temperature from a TMP102 I2C sensor.
5  //
6  // This software is provided under the CC BY-SA 3.0 licence. A
7  // copy of this licence can be found at:
8  //
9  // http://creativecommons.org/licenses/by-sa/3.0/legalcode
10 //
11 #if defined DISCOVERY
12     #include <iostm8s105c6.h>
13 #else
14     #include <iostm8s103f3.h>
15 #endif
16 #include <intrinsics.h>
17
18 //
19 // Define some pins to output diagnostic data.
20 //
21 #define PIN_BIT_BANG_DATA      PD_ODR_ODR4
22 #define PIN_BIT_BANG_CLOCK    PD_ODR_ODR5
23 #define PIN_ERROR              PD_ODR_ODR6
24
25 //
26 // I2C device related constants.
27 //
28 #define DEVICE_ADDRESS        0x48
29 #define I2C_READ               1
30 #define I2C_WRITE             0
31
32 //
33 // Buffer to hold the I2C data.
34 //
35 unsigned char _buffer[2];
36 int _nextByte = 0;
37
38 //
39 // Bit bang data on the diagnostic pins.
40 //
41 void BitBang(unsigned char byte)
42 {
43     for (short bit = 7; bit >= 0; bit--)
44     {
45         if (byte & (1 << bit))
46         {
47             PIN_BIT_BANG_DATA = 1;
48         }
49         else
50         {
51             PIN_BIT_BANG_DATA = 0;
52         }
53         PIN_BIT_BANG_CLOCK = 1;
54         __no_operation();
55     }
56 }
```

This website uses cookies to improve your experience and to gather page view statistics. This site does not collect user information. [Accept & Close](#) [Read More](#)

```

60 //
61 // Set up the system clock to run at 16MHz using the internal oscillator.
62 //
63 void InitialiseSystemClock()
64 {
65     CLK_IICKR = 0; // Reset the Internal Clock Register
66     CLK_IICKR_HSIEN = 1; // Enable the HSI.
67     CLK_EICKR = 0; // Disable the external clock.
68     while (CLK_IICKR_HSIIRDY == 0); // Wait for the HSI to be ready for
69     CLK_CKDIVR = 0; // Ensure the clocks are running at
70     CLK_PCKENR1 = 0xff; // Enable all peripheral clocks.
71     CLK_PCKENR2 = 0xff; // Ditto.
72     CLK_CCOR = 0; // Turn off CCO.
73     CLK_HSITRIMR = 0; // Turn off any HSIU trimming.
74     CLK_SWIMCCR = 0; // Set SWIM to run at clock / 2.
75     CLK_SWR = 0xe1; // Use HSI as the clock source.
76     CLK_SWCR = 0; // Reset the clock switch control register
77     CLK_SWCR_SWEN = 1; // Enable switching.
78     while (CLK_SWCR_SWBSY != 0); // Pause while the clock switch is
79 }
80
81 //
82 // Initialise the I2C system.
83 //
84 void InitialiseI2C()
85 {
86     I2C_CR1_PE = 0; // Disable I2C before configuration
87     //
88     // Setup the clock information.
89     //
90     I2C_FREQR = 16; // Set the internal clock frequency
91     I2C_CCRH_F_S = 0; // I2C running in standard mode.
92     I2C_CCRL = 0xa0; // SCL clock speed is 50 KHz.
93     I2C_CCRH_CCR = 0x00;
94     //
95     // Set the address of this device.
96     //
97     I2C_OARH_ADDMODE = 0; // 7 bit address mode.
98     I2C_OARH_ADDCONF = 1; // Docs say this must always be 1.
99     //
100    // Setup the bus characteristics.
101    //
102    I2C_TRISER = 17;
103    //
104    // Turn on the interrupts.
105    //
106    I2C_ITR_ITBUFEN = 1; // Buffer interrupt enabled.
107    I2C_ITR_ITEVTEN = 1; // Event interrupt enabled.
108    I2C_ITR_ITERREN = 1;
109    //
110    // Configuration complete so turn the peripheral on.
111    //
112    I2C_CR1_PE = 1;
113    //
114    // Enter master mode.
115    //

```

This website uses cookies to improve your experience and to gather page view statistics. This site does not collect user information. [Accept & Close](#) [Read More](#)

```
121 // I2C interrupts all share the same handler.
122 //
123 #pragma vector = I2C_RXNE_vector
124 __interrupt void I2C_IRQHandler()
125 {
126     if (I2C_SR1_SB)
127     {
128         //
129         // Master mode, send the address of the peripheral we
130         // are talking to. Reading SR1 clears the start condition.
131         //
132         reg = I2C_SR1;
133         //
134         // Send the slave address and the read bit.
135         //
136         I2C_DR = (DEVICE_ADDRESS << 1) | I2C_READ;
137         //
138         // Clear the address registers.
139         //
140         I2C_OARL_ADD = 0;
141         I2C_OARH_ADD = 0;
142         return;
143     }
144     if (I2C_SR1_ADDR)
145     {
146         //
147         // In master mode, the address has been sent to the slave.
148         // Clear the status registers and wait for some data from the slave
149         //
150         reg = I2C_SR1;
151         reg = I2C_SR3;
152         return;
153     }
154     if (I2C_SR1_RXNE)
155     {
156         //
157         // The TMP102 temperature sensor returns two bytes of data
158         //
159         _buffer[_nextByte++] = I2C_DR;
160         if (_nextByte == 1)
161         {
162             I2C_CR2_ACK = 0;
163             I2C_CR2_STOP = 1;
164         }
165         else
166         {
167             BitBang(_buffer[0]);
168             BitBang(_buffer[1]);
169         }
170         return;
171     }
172     //
173     // If we get here then we have an error so clear
174     // the error and continue.
175     //
176     unsigned char reg = I2C_SR1;
```

This website uses cookies to improve your experience and to gather page view statistics. This site does not collect user information. [Accept & Close](#) [Read More](#)

```

182     PIN_ERROR = 1;
183     __no_operation();
184     PIN_ERROR = 0;
185 }
186
187 //
188 // Main program loop.
189 //
190 int main()
191 {
192     __disable_interrupt();
193     //
194     // Initialise Port D.
195     //
196     PD_ODR = 0;           // All pins are turned off.
197     PD_DDR_DDR4 = 1;      // Port D, bit 4 is output.
198     PD_CR1_C14 = 1;       // Pin is set to Push-Pull mode.
199     PD_CR2_C24 = 1;       // Pin can run up to 10 MHz.
200     //
201     PD_DDR_DDR5 = 1;      // Port D, bit 5 is output.
202     PD_CR1_C15 = 1;       // Pin is set to Push-Pull mode.
203     PD_CR2_C25 = 1;       // Pin can run up to 10 MHz.
204     //
205     PD_DDR_DDR6 = 1;      // Port D, bit 6 is output.
206     PD_CR1_C16 = 1;       // Pin is set to Push-Pull mode.
207     PD_CR2_C26 = 1;       // Pin can run up to 10 MHz.
208     //
209     InitialiseSystemClock();
210     InitialiseI2C();
211     __enable_interrupt();
212     while (1)
213     {
214         __wait_for_interrupt();
215     }
216 }

```

Putting this in a project and running on the STM8S gives the following output on the Saleae logic analyser:



I2C Communication between STM8S and a TMP102

The output looks similar to that from the Netduino application above. Breaking out the calculator and using the readings in the above screen shot gives a temperature of 19.6 C which is right according to the thermometer in the room.

Conclusion

The above application shows the basics of a master I2C application. The code needs to be expanded to add some error handling to detect some of the errors that can occur (bus busy, acknowledge failures etc.) but the basics are there.

This website uses cookies to improve your experience and to gather page view statistics. This site does not collect user information. [Accept & Close](#) [Read More](#)

One Response to “STM8S I2C Master Devices”

1. [Silverlight Developer » Blog Archive STM8S I2C Slave Device - Silverlight Developer](#) says:
[May 24, 2015 at 9:33 am](#)

[...] succeeded at getting a basic I2C master working on the STM8S it is not time to start to look at I2C slave devices on the [...]

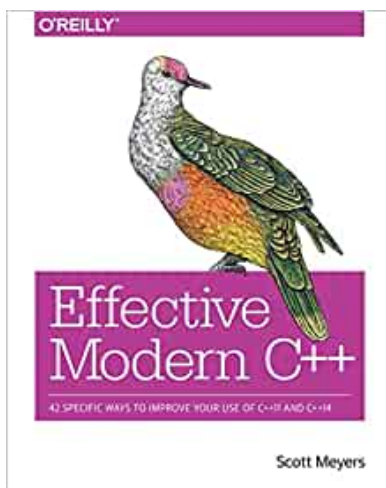
Pages

- [About](#)
- [Making a Netduino GO! Module](#)
- [The Way of the Register](#)
- [Making an IR Remote Control](#)
- [Weather Station Project](#)
- [NuttX and Raspberry Pi PicoW](#)

Support This Site



Currently Reading



© 2010 - 2024 Mark Stevens