Using hardware and software to make new stuff

Search

type, hit enter

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

- April 2023
- 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

- October 2011
- September 2011
- July 2011
- June 2011
- May 2011
- April 2011
- March 2011
- February 2011
- January 2011
- December 2010
- April 2010

STM8S I2C Slave Device

Having 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 STM8S.

The project will be a simple one, the STM8S will take a stream of bytes and perform simple addition. When requested, the STM8S will return the total as a 16-bit integer. Each new write request will clear the current total and start the whole process again.

Simple enough so let's get started.

This post is a fairly long as it contains a lot of code so it might be a good time to grab a beer and settle down.

Simple Slave Adder

The slave adder is a simple device running over I2C. The devices has the following properties:

1. Acts as an I2C slave device with address 0x50
2. I2C bus speed is 50 KHz
3. Upon receiving a valid start condition and address for a write operation the device will clear the current total.
4. Bytes written to the device will be summed and a running total kept.
5. A start condition with a valid address for a read operation will return the current total as a 16-bit integer, MSB first.

A Netduino 3 will be used as the I2C bus master device. This runs the .NET Microframework and has an implementation of the I2C protocol built into the framework. This gives a good reference point for the master device i.e. someone else has debugged that so we can assume that the device is working as per the protocol specification.

As mentioned in the previous article on I2C Master devices, there is a really good article about the I2C protocol on Wikipedia. If you want more information about the protocol then I suggest you head over there.

Netduino Code

The initial version of the I2C master device will simply send two bytes of data for the I2C slave device to sum. The code is simple enough:

```
1  using Microsoft.SPOT;
2  using Microsoft.SPOT.Hardware;
3  using System.Threading;
4
```

```
        public static void Main()
        {
            //
            //  Create a new I2C object on address 0x50 with the clock runnir
            //
            I2CDevice i2cBus = new I2CDevice(new I2CDevice.Configuration(0x50
            //
            //  Create a transaction to write two bytes of data to the I2C bu
            //
            byte[] buffer = { 1, 2 };
            I2CDevice.I2CTransaction[] transactions = new I2CDevice.I2CTransa
            transactions[0] = I2CDevice.CreateWriteTransaction(buffer);
            while (true)
            {
                int bytesRead = i2cBus.Execute(transactions, 100);
                Thread.Sleep(1000);
            }
        }
    }
}
```

The above application creates a new instance of the I2CDevice with a device address of)x50 and a clock frequency of 50 KHz. A single transaction is created and the master writes the same two bytes to the I2C bus every second.

STM8S Code

As with the I2C Master article, much of the action happens in two places:

1. I2C device initialisation method
2. I2C Interrupt Service Routine (ISR)

The initialisation method sets up the I2C peripheral on the STM8S and enters the waiting state, waiting for the master to put data onto the I2C bus.

The ISR will deal with the actual data processing and in a full application it will also deal with any error conditions that may arise.

I2C Initialisation

The initialisation method sets up the I2C bus by performing the following tasks:

1. Setting the expected clock frequency
2. Setting the device address
3. Turning the interrupts on

The I2C peripheral must be disabled before configuring certain aspects of the peripheral, namely clock speeds:

```
I2C_CR1_PE = 0;
```

The peripheral needs to know the current master clock frequency, the I2C mode (*Standard* or *Fast*) and the clock divider values. These must all be configured whilst the peripheral is disabled.

```
I2C_FREQR = 16;                 //  Set the internal clock frequency (MHz)
```

tutorials in this series. This is indicated by the *I2C_FREQR* register.

The values for the clock control register (*I2C_CCRL* and *I2C_CCRH_CCR* were simply taken from the STM8S programming reference manual. There is no coincidence that a bus speed of 50 KHz was chose, it is simply one of the reference values in the table. No point in creating work if you do not have to. If you want different speeds then you can either use one of the defined values in the remainder of the table or use the formulae provided.

The next step is to define the device address and addressing mode. I2C allows both 7 and 10 bit addressing modes. For simplicity this device will use a 7-bit device address:

```
1  I2C_OARH_ADDMODE = 0;                //  7 bit address mode.
2  I2C_OARH_ADD = 0;                    //  Set this device address to be 0x50.
3  I2C_OARL_ADD = 0x50;
4  I2C_OARH_ADDCONF = 1;                //  Docs say this must always be 1.
```

The device also allows for the maximum rise time to be configured. This example uses 17uS as the maximum time:

```
1  I2C_TRISER = 17;
```

The I2C peripheral allows for three different interrupt conditions to be defined, buffer interrupts, event interrupts (start condition etc.) and error interrupts. For convenience we will turn all of these on:

```
1  I2C_ITR_ITBUFEN = 1;                 //  Buffer interrupt enabled.
2  I2C_ITR_ITEVTEN = 1;                 //  Event interrupt enabled.
3  I2C_ITR_ITERREN = 1;
```

Now that the peripheral is configured we need to re-enable it:

```
1  I2C_CR1_PE = 1;
```

The last bit of initialisation is to configure the device to return an ACK after each byte:

```
1  I2C_CR2_ACK = 1;
```

At this point the I2C peripheral should be listening to the I2C bus for a start condition and the address 0x50.

I2C Interrupt Service Routine (ISR)

The ISR contains the code which processes the data and error conditions for the I2C peripheral. All of the I2C events share the same ISR and the ISR will need to interrogate the status registers in order to determine the exact reason for the interrupt.

Starting with an empty ISR we have the following code:

```
1  #pragma vector = I2C_RXNE_vector
2  __interrupt void I2C_IRQHandler()
3  {
4  }
```

generated.

1. Address detection
2. Receive buffer not empty

This initial simple implementation can use these two events to clear the total when a new Start condition is received and add the current byte to the total when data is received.

```
if (I2C_SR1_ADDR)
{
    //
    //  In master mode, the address has been sent to the slave.
    //  Clear the status registers and wait for some data from the salve.
    //
    reg = I2C_SR1;
    reg = I2C_SR3;
    _total = 0;                 // New addition so clear the total.
    return;
}
```

I2C_SR1_ADDR should be set when an address is detected on the bus which matches the address currently in the address registers. As a starter application the code can simply assume that any address is going to be a write condition. The code can clear the totals and get ready to receive data. Note that this will be expanded later to take into consideration the fact that the master can perform both read and write operations.

The next event we need to consider is the receipt of data from the master. This will trigger the *Receive Buffer Not Empty* interrupt. This simple application should just read the buffer and add the current byte to the running total:

```
if (I2C_SR1_RXNE)
{
    //
    //  Received a new byte of data so add to the running total.
    //
    _total += I2C_DR;
    return;
}
```

As indicated earlier, the I2C ISR is a generic ISRT and is triggered for all I2C events. The initialisation code above has turned on the error interrupts as well as the data capture interrupts. Whilst none of the code in this article will handle error conditions, the ISR will output the status registers for diagnostic purposes:

```
PIN_ERROR = 1;
__no_operation();
PIN_ERROR = 0;
reg = I2C_SR1;
BitBang(reg);
reg = I2C_SR3;
BitBang(reg);
```

This will of course require suitable definitions and support methods.

Putting this all together gives an initial implementation of a slave device as:

```
//
```

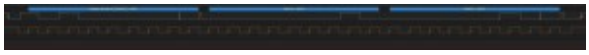
```
//
//  This software is provided under the CC BY-SA 3.0 licence.  A
//  copy of this licence can be found at:
//
//  http://creativecommons.org/licenses/by-sa/3.0/legalcode
//
#if defined DISCOVERY
    #include <iostm8S105c6.h>
#else
    #include <iostm8s103f3.h>
#endif
#include <intrinsics.h>

//
//  Define some pins to output diagnostic data.
//
#define PIN_BIT_BANG_DATA       PD_ODR_ODR4
#define PIN_BIT_BANG_CLOCK      PD_ODR_ODR5
#define PIN_ERROR               PD_ODR_ODR6

//
//  Somewhere to hold the sum.
//
int _total;

//
//  Bit bang data on the diagnostic pins.
//
void BitBang(unsigned char byte)
{
    for (short bit = 7; bit >= 0; bit--)
    {
        if (byte & (1 << bit))
        {
            PIN_BIT_BANG_DATA = 1;
        }
        else
        {
            PIN_BIT_BANG_DATA = 0;
        }
        PIN_BIT_BANG_CLOCK = 1;
        __no_operation();
        PIN_BIT_BANG_CLOCK = 0;
    }
    PIN_BIT_BANG_DATA = 0;
}

//
//  Set up the system clock to run at 16MHz using the internal oscillator.
//
void InitialiseSystemClock()
{
    CLK_ICKR = 0;                       //  Reset the Internal Clock Registe
    CLK_ICKR_HSIEN = 1;                 //  Enable the HSI.
    CLK_ECKR = 0;                       //  Disable the external clock.
    while (CLK_ICKR_HSIRDY == 0);       //  Wait for the HSI to be ready for
    CLK_CKDIVR = 0;                     //  Ensure the clocks are running at
```

```
    CLK_SWR = 0xe1;                     //  Use HSI as the clock source.
    CLK_SWCR = 0;                       //  Reset the clock switch control r
    CLK_SWCR_SWEN = 1;                  //  Enable switching.
    while (CLK_SWCR_SWBSY != 0);        //  Pause while the clock switch is
}

//
//  Initialise the I2C system.
//
void InitialiseI2C()
{
    I2C_CR1_PE = 0;                     //  Disable I2C before configuratior
    //
    //  Set up the clock information.
    //
    I2C_FREQR = 16;                     //  Set the internal clock frequency
    I2C_CCRH_F_S = 0;                   //  I2C running is standard mode.
    I2C_CCRL = 0xa0;                    //  SCL clock speed is 50 KHz.
    I2C_CCRH_CCR = 0x00;
    //
    //  Set the address of this device.
    //
    I2C_OARH_ADDMODE = 0;               //  7 bit address mode.
    I2C_OARH_ADD = 0;                   //  Set this device address to be 0x
    I2C_OARL_ADD = 0x50;
    I2C_OARH_ADDCONF = 1;               //  Docs say this must always be 1.
    //
    //  Set up the bus characteristics.
    //
    I2C_TRISER = 17;
    //
    //  Turn on the interrupts.
    //
    I2C_ITR_ITBUFEN = 1;                //  Buffer interrupt enabled.
    I2C_ITR_ITEVTEN = 1;                //  Event interrupt enabled.
    I2C_ITR_ITERREN = 1;
    //
    //  Configuration complete so turn the peripheral on.
    //
    I2C_CR1_PE = 1;
    //
    //  Acknowledge each byte with an ACK signal.
    //
    I2C_CR2_ACK = 1;
}

//
//  I2C interrupts all share the same handler.
//
#pragma vector = I2C_RXNE_vector
__interrupt void I2C_IRQHandler()
{
    unsigned char reg;

    if (I2C_SR1_ADDR)
    {
```

```
            _total = 0;                 // New addition so clear the total.
            return;
        }
        if (I2C_SR1_RXNE)
        {
            //
            //  Received a new byte of data so add to the running total.
            //
            _total += I2C_DR;
            return;
        }
        //
        //  Send a diagnostic signal to indicate we have cleared
        //  the error condition.
        //
        PIN_ERROR = 1;
        __no_operation();
        PIN_ERROR = 0;
        //
        //  If we get here then we have an error so clear
        //  the error, output the status registers and continue.
        //
        reg = I2C_SR1;
        BitBang(reg);
        reg = I2C_SR3;
        BitBang(reg);
    }

    //
    //  Main program loop.
    //
    int main()
    {
        _total = 0;
        __disable_interrupt();
        //
        //  Initialise Port D.
        //
        PD_ODR = 0;             //  All pins are turned off.
        PD_DDR_DDR4 = 1;        //  Port D, bit 4 is output.
        PD_CR1_C14 = 1;         //  Pin is set to Push-Pull mode.
        PD_CR2_C24 = 1;         //  Pin can run up to 10 MHz.
        //
        PD_DDR_DDR5 = 1;        //  Port D, bit 5 is output.
        PD_CR1_C15 = 1;         //  Pin is set to Push-Pull mode.
        PD_CR2_C25 = 1;         //  Pin can run up to 10 MHz.
        //
        PD_DDR_DDR6 = 1;        //  Port D, bit 6 is output.
        PD_CR1_C16 = 1;         //  Pin is set to Push-Pull mode.
        PD_CR2_C26 = 1;         //  Pin can run up to 10 MHz.
        //
        InitialiseSystemClock();
        InitialiseI2C();
        __enable_interrupt();
        while (1)
        {
```

Executing the Applications

At this stage a quick test with the two devices connected and the logic analyser will show the master device outputting data to the I2C bus. The correct operation of the I2C salve device can be verified by setting break points with in the two if statements in the ISR. Note that this will generate some errors but it is good enough to verify that the ISRs are being triggered correctly.



STM8 I2C Write Condition

The above shows the output from the Saleae Logic Analyser when the write transaction is sent to the bus by the Netduino.

Reading and Writing with I2C Slaves

At this point the above code should be accepting data from the Netduino 3. It is now time to expand the code to take into account the requirement to read back data from the slave device.

Netduino Code

The Netduino code will need to be modified to generate both a write transaction to send data to the I2C slave device and a read transaction to retrieve the results from the calculation.

```
using Microsoft.SPOT;
using Microsoft.SPOT.Hardware;
using System.Threading;

namespace I2CMaster
{
    public class Program
    {
        public static void Main()
        {
            //
            //  Create a new I2C object on address 0x50 with the clock runnir
            //
            I2CDevice i2cBus = new I2CDevice(new I2CDevice.Configuration(0x50
            //
            //  Create a transaction to write two bytes of data to the I2C bu
            //
            byte[] buffer = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
            byte[] resultBuffer = new byte[2];
            I2CDevice.I2CTransaction[] transactions = new I2CDevice.I2CTransa
            transactions[0] = I2CDevice.CreateWriteTransaction(buffer);
            transactions[1] = I2CDevice.CreateReadTransaction(resultBuffer);
            while (true)
            {
                int bytesRead = i2cBus.Execute(transactions, 100);
                Thread.Sleep(1000);
            }
        }
    }
```

data to be summed has also been expanded to include a further eight elements.

STM8S Code

A state machine will be used to allow the STM8S to work out what action should be taken within the ISR. This is required as the i”c peripheral will generate an event for the address detection for both the write transaction and the read transaction. The application will need to be able to differentiate between the two conditions as one requires the total to be cleared, the other requires that the total is sent back to the master device.

The state machine contains the following states:

1. Waiting for a write condition to start
2. Adding data to the running total
3. Sending the MSB of the total
4. Sending the LSB of the total

This is represented in code by an enum and a global variable:

```
typedef enum
{
    ISWaiting,                  //  Waiting for comms to start.
    ISAdding,                   //  Adding bytes of data.
    ISSendingMSB,               //  Sending MSB of total.
    ISSendingLSB                //  Sending LSB of total.
} I2CStateType;
I2CStateType _i2cState;
```

The ISR will need to be changed in order to deal with the state machine and also handle the slave transmitting data to the master device. The first change is to the address detection. This assumes that the main program loop initialises the state to *ISWaiting*:

```
if (I2C_SR1_ADDR)
{
    //
    //  Slave address received, work out if to expect a read or a write.
    //
    if (_i2cState == ISWaiting)
    {
        _i2cState = ISAdding;
        _total = 0;             // New addition so clear the total.
    }
    reg = I2C_SR1;
    reg = I2C_SR3;
    return;
}
```

The code above works out if the application is waiting for the first write condition (*IsWaiting*) or if the address detection has been triggered by a read condition (any other state).

Next, any other data reception is assumed to be data to be added to the total. The state is changed to *IsAdding* just in case:

```
if (I2C_SR1_RXNE)
{
```

```
        return;
    }
```

Next we have two new conditions we have not considered so far, the first is the fact that we need to transmit the result to the master. The request from the client will trigger a *Transmit Buffer Empty* event:

```
if (I2C_SR1_TXE)
{
    if (_i2cState == ISAdding)
    {
        I2C_DR = (_total >> 8) & 0xff;
        _i2cState = ISSendingMSB;
    }
    else
    {
        I2C_DR = _total & 0xff;
        _i2cState = ISSendingLSB;
    }
    return;
}
```

The master is asking for two bytes and the application needs to track if we are transmitting the most significant byte (MSB) or least significant byte (LSB).

The next new condition is the acknowledge event. This is generated at the end of the master read operation. This will set the system back into a waiting condition:

```
if (I2C_SR2_AF)
{
    I2C_SR2_AF = 0;             //  End of slave transmission.
    _i2cState = ISWaiting;
    return;
}
```

The full application becomes:

```
//
//  This application demonstrates the principles behind developing an
//  I2C slave device on the STM8S microcontroller.  The application
//  will total the byte values written to it and then send the total
//  to the master when the device is read.
//
//  This software is provided under the CC BY-SA 3.0 licence.  A
//  copy of this licence can be found at:
//
//  http://creativecommons.org/licenses/by-sa/3.0/legalcode
//
#if defined DISCOVERY
    #include <iostm8S105c6.h>
#else
    #include <iostm8s103f3.h>
#endif
#include <intrinsics.h>

```

```
#define PIN_ERROR               PD_ODR_ODR6

//
//  State machine for the I2C communications.
//
typedef enum
{
    ISWaiting,                  //  Waiting for comms to start.
    ISAdding,                   //  Adding bytes of data.
    ISSendingMSB,               //  Sending MSB of total.
    ISSendingLSB                //  Sending LSB of total.
} I2CStateType;
I2CStateType _i2cState;

//
//  Somewhere to hold the sum.
//
int _total;

//
//  Bit bang data on the diagnostic pins.
//
void BitBang(unsigned char byte)
{
    for (short bit = 7; bit >= 0; bit--)
    {
        if (byte & (1 << bit))
        {
            PIN_BIT_BANG_DATA = 1;
        }
        else
        {
            PIN_BIT_BANG_DATA = 0;
        }
        PIN_BIT_BANG_CLOCK = 1;
        __no_operation();
        PIN_BIT_BANG_CLOCK = 0;
    }
    PIN_BIT_BANG_DATA = 0;
}

//
//  Set up the system clock to run at 16MHz using the internal oscillator.
//
void InitialiseSystemClock()
{
    CLK_ICKR = 0;                       //  Reset the Internal Clock Registe
    CLK_ICKR_HSIEN = 1;                 //  Enable the HSI.
    CLK_ECKR = 0;                       //  Disable the external clock.
    while (CLK_ICKR_HSIRDY == 0);       //  Wait for the HSI to be ready for
    CLK_CKDIVR = 0;                     //  Ensure the clocks are running at
    CLK_PCKENR1 = 0xff;                 //  Enable all peripheral clocks.
    CLK_PCKENR2 = 0xff;                 //  Ditto.
    CLK_CCOR = 0;                       //  Turn off CCO.
    CLK_HSITRIMR = 0;                   //  Turn off any HSIU trimming.
    CLK_SWIMCCR = 0;                    //  Set SWIM to run at clock / 2.
```

```
 85
 86  //
 87  //  Set up Port D GPIO for diagnostics.
 88  //
 89  void InitialisePortD()
 90  {
 91      PD_ODR = 0;             //  All pins are turned off.
 92      PD_DDR_DDR4 = 1;        //  Port D, bit 4 is output.
 93      PD_CR1_C14 = 1;         //  Pin is set to Push-Pull mode.
 94      PD_CR2_C24 = 1;         //  Pin can run up to 10 MHz.
 95      //
 96      PD_DDR_DDR5 = 1;        //  Port D, bit 5 is output.
 97      PD_CR1_C15 = 1;         //  Pin is set to Push-Pull mode.
 98      PD_CR2_C25 = 1;         //  Pin can run up to 10 MHz.
 99      //
100      PD_DDR_DDR6 = 1;        //  Port D, bit 6 is output.
101      PD_CR1_C16 = 1;         //  Pin is set to Push-Pull mode.
102      PD_CR2_C26 = 1;         //  Pin can run up to 10 MHz.
103  }
104
105  //
106  //  Initialise the I2C system.
107  //
108  void InitialiseI2C()
109  {
110      I2C_CR1_PE = 0;                     //  Disable I2C before configuratior
111      //
112      //  Set up the clock information.
113      //
114      I2C_FREQR = 16;                     //  Set the internal clock frequency
115      I2C_CCRH_F_S = 0;                   //  I2C running is standard mode.
116      I2C_CCRL = 0xa0;                    //  SCL clock speed is 50 KHz.
117      I2C_CCRH_CCR = 0x00;
118      //
119      //  Set the address of this device.
120      //
121      I2C_OARH_ADDMODE = 0;               //  7 bit address mode.
122      I2C_OARH_ADD = 0;                   //  Set this device address to be 0x
123      I2C_OARL_ADD = 0x50;
124      I2C_OARH_ADDCONF = 1;               //  Docs say this must always be 1.
125      //
126      //  Set up the bus characteristics.
127      //
128      I2C_TRISER = 17;
129      //
130      //  Turn on the interrupts.
131      //
132      I2C_ITR_ITBUFEN = 1;                //  Buffer interrupt enabled.
133      I2C_ITR_ITEVTEN = 1;                //  Event interrupt enabled.
134      I2C_ITR_ITERREN = 1;
135      //
136      //  Configuration complete so turn the peripheral on.
137      //
138      I2C_CR1_PE = 1;
139      //
140      //  Set the acknowledge to be ACK.
```

```
//
//  I2C interrupts all share the same handler.
//
#pragma vector = I2C_RXNE_vector
__interrupt void I2C_IRQHandler()
{
    unsigned char reg;

    if (I2C_SR1_ADDR)
    {
        //
        //  Slave address received, work out if to expect a read or a write.
        //
        if (_i2cState == ISWaiting)
        {
            _i2cState = ISAdding;
            _total = 0;             // New addition so clear the total.
        }
        reg = I2C_SR1;
        reg = I2C_SR3;
        return;
    }
    if (I2C_SR1_RXNE)
    {
        //
        //  Receiving data from the master so we must be adding.
        //
        _total += I2C_DR;
        _i2cState = ISAdding;
        return;
    }
    if (I2C_SR1_TXE)
    {
        if (_i2cState == ISAdding)
        {
            I2C_DR = (_total >> 8) & 0xff;
            _i2cState = ISSendingMSB;
        }
        else
        {
            I2C_DR = _total & 0xff;
            _i2cState = ISSendingLSB;
        }
        return;
    }
    if (I2C_SR2_AF)
    {
        I2C_SR2_AF = 0;             //  End of slave transmission.
        _i2cState = ISWaiting;
        return;
    }
    //
    //  Send a diagnostic signal to indicate we have cleared
    //  the error condition.
    //
    PIN_ERROR = 1;
    __no_operation();
```

```
207     reg = I2C_SR1;
208     BitBang(reg);
209     BitBang(I2C_SR2);
210     reg = I2C_SR3;
211     BitBang(reg);
212 }
213
214 //
215 //  Main program loop.
216 //
217 int main()
218 {
219     _total = 0;
220     _i2cState = ISWaiting;
221     __disable_interrupt();
222     InitialisePortD();
223     InitialiseSystemClock();
224     InitialiseI2C();
225     __enable_interrupt();
226     while (1)
227     {
228         __wait_for_interrupt();
229     }
230 }
```

The above two pieces of code are fuller applications and if we execute these and hook up the logic anayser to the I2C bus we see the following output:

Adding 1 To 10

Reading From Multiple Devices

The I2C protocol allows for one *or more* devices with different slave addresses to be connected to the same I2C bus. The previous article used the TMP102 temperature sensor with slave address 0x48 and this article has created a slave device with address 0x50. It should therefore be possible to connect the two devices to the same bus and talk to each selectively.

Netduino Application

As with the previous examples, the Netduino 3 will be used as the I2C bus master. The applicaiton above will need to be merged with the code in the previous article.

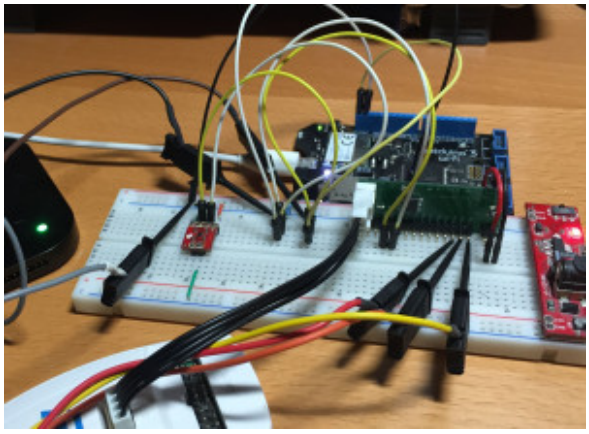
```
1 using Microsoft.SPOT;
2 using Microsoft.SPOT.Hardware;
3 using System.Threading;
4
5 namespace I2CMaster
6 {
7     public class Program
```

```
            //  Create a new I2C object and the configurations for the STM8S
            //
            I2CDevice.Configuration stm8s = new I2CDevice.Configuration(0x50,
            I2CDevice.Configuration tmp102 = new I2CDevice.Configuration(0x48
            I2CDevice i2cBus = new I2CDevice(stm8s);
            //
            //  Create a transaction to write several bytes of data to the I2
            //
            byte[] buffer = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
            byte[] resultBuffer = new byte[2];
            I2CDevice.I2CTransaction[] transactions = new I2CDevice.I2CTransa
            transactions[0] = I2CDevice.CreateWriteTransaction(buffer);
            transactions[1] = I2CDevice.CreateReadTransaction(resultBuffer);
            //
            //  Create a transaction to read two bytes of data from the TMP10
            //
            byte[] temperatureBuffer = new byte[2];
            I2CDevice.I2CTransaction[] reading = new I2CDevice.I2CTransactior
            reading[0] = I2CDevice.CreateReadTransaction(temperatureBuffer);
            while (true)
            {
                //
                //  Read data from the I2C bus.
                //
                i2cBus.Config = stm8s;
                int bytesRead = i2cBus.Execute(transactions, 100);
                i2cBus.Config = tmp102;
                bytesRead = i2cBus.Execute(reading, 100);
                //
                //  Convert the reading into Centigrade and Fahrenheit.
                //
                int sensorReading = ((temperatureBuffer[0] << 8) | temperatur
                double centigrade = sensorReading * 0.0625;
                double fahrenheit = centigrade * 1.8 + 32;
                //
                //  Display the readings in the debug window and pause before
                //
                Debug.Print(centigrade.ToString() + " C / " + fahrenheit.ToSt
                //
                //  Now display the results of the addition.
                //
                string message = "";
                for (int index = 0; index < buffer.Length; index++)
                {
                    message += buffer[index].ToString();
                    if (index == (buffer.Length - 1))
                    {
                        message += " = " + ((resultBuffer[0] * 256) + resultE
                    }
                    else
                    {
                        message += " + ";
                    }
                }
                Debug.Print(message);
                Thread.Sleep(1000);
            }
```

The above code creates two I2C configuration object, one for the TMP102 and one for the STM8S device. The I2C bus object has the configuration changed depending upon which device is required.

Wiring up the Devices

Firstly, wire up the TMP102 sensor as described in the article on I2C master devices. The SDA and SCK lines should be connected to 3.3V via 4K7 resistors. Next connect the SDA and SCK lines from the STM8S device to the same point as SDA and SCK lines from the TMP102 sensor. Throw in the logic analyser connections and you get something like this:



TMP102 and STM8S I2C slave devices

Running the Application

Running the above two applications and starting the logic analyser generated the following output:

Reading From Two I2C Slaves

The write operation to the far left (W0x50) sends the 10 bytes to the STM8S device. This is followed by a read (R0x50) of two bytes from the same device, this can be seen about two thirds of the way from the left-hand side of the trace. The final operation is the read of the current temperature, R0x48, to the right of the trace.

Conclusion

The STM8S slave device above is a simple device. There are still plenty of modification which need to be made:

1. Add error handling
2. Allow for trapping conditions such as clock stretching

to name but a few. The code above should allow for simple devices to be put together is a short space of time.

Tags: Electronics, Netduino, Software Development, STM8

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

Buy me a coffee

Currently Reading