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

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

STM8S SPI Slave (Part 3) – Making a Go Module

In this, the last of the series of posts regarding implementing SPI Slave devices using the STM8S, we will look at building a module for the Netduino Go. This post builds upon the two previous posts:

- STM8S SPI Slave
  A simple SPI implementation using hardware SPI (single byte exchange).
- STM8S SPI Slave (Part 2)
  Software SPI implementation (using buffers to allow the exchange of data packets).

Here we will build upon the buffering and overlay the the Netduino Go 1.0 protocol in order to allow the two devices to communicate. We will also extend the STM8S application to add a simple function table to allow the simple addition of extra functionality to the system.

The makers of the Netduino Go, Secret Labs, have not formally released the GoBus 1.0 specification as a document. They have however release the source code to some of their modules and this can be found in the Wiki. The code found in the Wiki posts along with discussions on various forums has been used in the production of the code presented here. Credit for help is due to Secret Labs for releasing the code and also to the following Netduino forum members:

- Chris Walker
- CW2

These forum members have given assistance in one form or another over the past few months and without their help this post would not have been possible.

GoBus 1.0 Protocol

The early GoBus protocol uses an 18 byte data packet which is exchanged by the Netduino Go and the module. This packet of data contains a one byte header, 16 bytes of data and a one byte checksum with the data packets being exchanged over SPI. With the exception of the header and the checksum it appears that meaning of the data within the 16 byte payload is determined by the module developer.

I would also point the reader to the blog post *A Developers Introduction to GoBus* by Matt Isenhower on the Komodex System web site.

Enumeration

When the Netduino Go is first powered it will look at each of the Go Sockets in turn and attempt to interrogate the modules which are connected to the main board. It does this by sending out a packet with a single byte header 0xfe followed by 16 data bytes and a checksum. From experience, the data bytes are normally set to 0xff.

board along with the corresponding socket numbers.

This process then allows the .NET code to connect to a module using the GUID or the GUID and socket number. Successful connection is indicated by the blue LED by the side of the socket being illuminated. A failed connection normally results in an exception being thrown.

Data Transfer/Module Control

When the code running on the Netduino Go has successfully attached to the module on a socket it can start to control/communicate with the module. At this point it appears that the protocol uses the header 0x80 to indicate transfer between the module and the main board. So our data packets remain 18 bytes with the following format:

- 0x80 – Header
- 16 byte payload
- 1 byte CRC

It appears that the meaning of the 16 byte payload is determined by the module developer.

GPIO Interrupt

The protocol also allows for the use of a single GPIO. This can be used as a signalling line to let either side know that an action is pending. Convention appears to be to use this to allow the module to let the code on the main board know that there is some data/action pending.

A Simple Module

We will be creating a simple module to illustrate how the STM8S and the Netduino code work together. In order to use the least hardware possible the module will perform a simple calculation and return the result. Our module will need to perform the following:

- Enumerate using a GUID allowing the Netduino Go to detect the module
- Receive a number from the Netduino Go
- Perform a simple calculation and notify the Netduino Go that the result is ready.
- Send the result back to the Netduino Go when requested.

This simple module illustrates the key types of communication which may be required of a simple module. It is of course possible to use these to perform other tasks such as controlling a LED or receiving input from a button or keypad.

Netduino Go Module Driver

The Netduino Go code derived from the C# code published by Secret Labs in their Wiki. The major changes which have been made for this post are really concerned with improving clarity (adding comments at each stage to expand on the key points etc.).

Module ID

Modules are identified using a GUID. This ID allows the GoBus to connect to a module by scanning the bus for the specified ID. It also allows the Netduino Go to verify that when connecting to a module on a specific socket that the module is of the correct type. So the first thing we will need to do is obtain a new GUID. There are various ways in which we can do this and the simplest way to do this is to use the *Create GUID* menu option in Visual Studio. You can find this in the *Tools* menu.

0x9e, 0x48, 0xd3, 0xfd, 0xae, 0x8c, 0xee });

REMEMBER: It is critical that you generate your own GUID as each module type will need to have distinct ID.

Scanning down the file a little way you will find the two constructors for the class. One takes a socket and attempts to bind to the specified module on the requested socket. The other will attach to the first available module on the GoBus.

Initialise

This method is key to allowing the Netduino Go to connect to the module. The method binds to the module (assuming the IDs match) and retrieves a list of resources which the driver can use to communicate with the module. In this case, the SPI information and the pin used as an interrupt port. The remainder of the method configures the module driver to use these resources.

One key point to note is the use of the *AutoResetEvent* object. This is used to allow the interrupt handler to communicate the fact that an event has occurred to the methods we will write. This can be done in a manner which is non-blocking.

AddFive Method

This is the first of our methods implementing the functionality which our module will provide. In our case, this method actually implements the simple arithmetic we will be asking the module to perform. We will be sending a byte to the module, the module will add five to the number passed and then make this available to the Netduino Go. The code looks like this:

```
1  public byte AddFive(byte value)
2  {
3      int retriesLeft = 10;
4      bool responseReceived = false;
5
6      _writeFrameBuffer[0] = GO_BUS10_COMMAND_RESPONSE;
7      _writeFrameBuffer[1] = CMD_ADD_FIVE;
8      _writeFrameBuffer[2] = value;
9      WriteDataToModule();
10     while (!responseReceived && (retriesLeft > 0))
11     {
12         //
13         //  We have written the data to the module so wait for a maximum
14         //  of 5 milliseconds to see if the module responds with some
15         //  data for us.
16         //
17         responseReceived = _irqPortInterruptEvent.WaitOne(5, false);
18         if ((responseReceived) && (_readFrameBuffer[1] == GO_BUS10_COMMAND_RE
19         {
20             //
21             //  Module has responded so extract the result.  Note we should r
22             //  verify the checksum at this point.
23             //
24             _writeFrameBuffer[0] = GO_BUS10_COMMAND_RESPONSE;
25             _writeFrameBuffer[1] = CMD_GET_RESULT;
26             WriteDataToModule();
27             return(_readFrameBuffer[2]);
28         }
29         else
```

```
            if (retriesLeft > 0)
            {
                WriteDataToModule();
            }
        }
    }
    throw new Exception("AddFive cannot communicate with the Basic GO! module
}
```

The first thing this the method does is to set up the *_writeFrameBuffer* with the header, the command number and the data we will be sending. The data is then written to the module using SPI.

Next we will wait a while for the module to indicate via the GPIO pin that it has processed the data and the result is ready. As we shall see later, the module has already put the result in the transmission buffer ready for retrieval. This will have been performed before the interrupt was generated. The following line performs the non-blocking check to see if the interrupt has been generated:

```
responseReceived = _irqPortInterruptEvent.WaitOne(5, false);
```

responseReceived will be true if the interrupt has been generated and the C# module code has received the event.

The final task is to retrieve the result from the module by sending a retrieve command. This is performed by the following code:

```
_writeFrameBuffer[0] = GO_BUS10_COMMAND_RESPONSE;
_writeFrameBuffer[1] = CMD_GET_RESULT;
WriteDataToModule();
return(_readFrameBuffer[2]);
```

STM8S Module

Much of the code required here has already been covered in the previous post, STM8S SPI Slave (Part 2). The protocol uses a small buffer to allow messages to be transferred between the STM8S and the Netduino Go. In buffers and provide a mechanism for interpreting these messages. The mechanism we adopted is as follows:

- All messages will be restricted to 18 bytes (one byte header, 16 bytes payload, one byte CRC)
- The request header (from the Netduino to the module) will be 0x80 allowing a 16 byte payload
- The response header (from the module to the Netduino) will be 0x2a followed by 0x80. This restricts the return payload to 15 bytes.
- The final byte will be a CRC calculated on the header and the payload

> The way in which the protocol has been implemented here also places a restriction upon on the application. Firstly, the module must receive a request as a full payload. Only then can the module respond. This is where the GPIO interrupt discussed earlier comes into play.
>
> The final part of the problem is to work out how to dispatch the messages received by the module. To do this we will use a function table.
>
> For the remainder of this article we will restrict ourselves to looking at the new functionality we will be adding on top of the previous post in order to allow the creation of a module.

following code allows the table to be setup:

```
//
//  Function table structure.
//
typedef struct
{
    unsigned char command;          //  Command number.
    void (*functionPointer)();      //  Pointer to the function to b
} FunctionTableEntry;
//
//  Forward function declarations for the function table.
//
void AddFive();
void GetValue();
//
//  Table of pointers to functions which implement the specified com
//
FunctionTableEntry _functionTable[] = { { 0x01, AddFive }, { 0x02, G
//
//  Number of functions in the function table.
//
const int _numberOfFunctions = sizeof(_functionTable) / sizeof(Funct
```

Here we define a function table entry which has a byte ID and a pointer to a function (taking a void parameter list) associated with the ID. We then declare an array of these objects and associate functions with the IDs.

The final line of code simply determines the number of entries in the function table.

Using the above table we can work out which function to call using the following code:

```
if (_numberOfFunctions > 0)
{
    for (int index = 0; index < _numberOfFunctions; index++)
    {
        if (_functionTable[index].command == _rxBuffer[1])
        {
            (*(_functionTable[index].functionPointer))();
            break;
        }
    }
}
```

The function table method presented here allows the functionality of the module to be expanded with relative ease. In order to add a new piece of functionality you simply need to do the following:

- Create a new method in the STM8S code to implement the new functionality
- Determine the ID to be used for the functionality and add a new entry to the function table
- Create a method in the Netduino Go driver to call the method and retrieve any results as necessary

By performing these three simple steps you can add one or more functions with ease. The communication protocol will continue to work as is with no modification. The only exception to this

We will need to make a slight modification to the Rx buffer in order to account for the checksum byte. We will also need to add somewhere to store the GUID which acts as the identifier for this module. This results in the following small change to the global variable code:

```
//
//  Application global variables.
//
unsigned char _rxBuffer[GO_BUFFER_SIZE + 1];    // Buffer holding th
unsigned char _txBuffer[GO_BUFFER_SIZE];        // Buffer holding th
unsigned char *_rx;                             // Place to put the
unsigned char *_tx;                             // Next byte to send
int _rxCount;                                   // Number of charact
int _txCount;                                   // Number of charact
volatile int _status;                           // Application statu
//
//  GUID which identifies this module.
//
unsigned char _moduleID[] = { 0x80, 0x39, 0xe8, 0x2b, 0x55, 0x58, 0x
                              0xab, 0x9e, 0x48, 0xd3, 0xfd, 0xae, 0x
```

GoBus Interrupt

The discussion of the code on the Netduino Go driver (on the main board) mentioned the fact that the module can raise an interrupt to signal the fact that an operation has completed and that data is ready for retrieval. In order to do this we raise an interrupt on one of the pins when we have processed the data. This code is trivial:

```
//
//  Raise an interrupt to the GO! main board to indicate that there
//  ready for collection.  The IRQ on the GO! board is configured as
//
//  _irqPort = new InterruptPort((Cpu.Pin) socketGpioPin, false, Por
//                               Port.InterruptMode.InterruptEdgeLow
//
void NotifyGOBoard()
{
    PIN_GOBUS_INTERRUPT = 0;
    __no_operation();
    PIN_GOBUS_INTERRUPT = 1;
}
```

This method is simple and really just toggles which is connected to GPIO pin on the Netduino Go socket.

Adding Functionality to the Module

In our simple case we need to add two pieces of functionality, the ability to add five to a number and then to allow the caller to retrieve the result. This results in the following two methods:

```
//
//  GO! function 1 - add 5 to byte 2 in the Rx buffer and put the ar
```

```
    NotifyGOBoard();
}

//-------------------------------------------------------------------
//
//  GO! Function 2 - return the Tx buffer back to the GO! board.
//
void GetValue()
{
    NotifyGOBoard();
}
```

SPI Go Frame

The implementation of the SPI processing here is interrupt driven. As such we will need to allow a method of synchronising the payloads we receive. This application will do this using the rising edge of the chip select signal which is generated by the Netduino Go main board. This allows us for cater for the many scenarios (synchronisation, underflow and overflow).

In the case of underflow and synchronisation, the chip select signal will rise before we have enough data. In this case we have either a corrupt packet or we have started to recei8ve data part way through the packet. In this case we cannot sensibly process the data so we should throw away the packet and wait for the next one.

An overflow situation can occur when the Netduino Go sends more than 18 bytes in one packet of data. In this case we should detect this and prevent the buffers from overflowing.

In order to allow for these cases we reset the Go frame pointers when the chip select signal changes from low to high:

```
//
//  This method resets SPI ready for the next transmission/receptior
//  on the GO! bus.
//
//  Do not call this method whilst SPI is enabled as it will have nc
//
void ResetGoFrame()
{
    if (!SPI_CR1_SPE)
    {
        (void) SPI_DR;                          //  Reset any error
        (void) SPI_SR;
        SPI_DR = GO_FRAME_PREFIX;               //  First byte of th
        _txBuffer[0] = _moduleID[0];            //  Second byte in t
        //
        //  Now reset the buffer pointers and counters ready for dat
        //
        _rx = _rxBuffer;
        _tx = _txBuffer;
        _rxCount = 0;
        _txCount = 0;
        //
        //  Note the documentation states this should be SPI_CR2_CRC
        //  but the header files have SPI_CR_CECEN defined.
```

```
        }
    }
```

As we shall see later, the end of the SPI transmission with result in one of the following cases:

- Too little data received correctly. The rising chip select line will reset the buffer pointers and the data will be discarded.
- The correct amount of data received. In this case the buffer will be processed correctly.
- Too much data is received. The excess data will be discarded to prevent a buffer overflow.

The *ResetGoFrame* method is key in ensuring that the buffers are reset at the end of the SPI transmission indicated by the rising chip select line.

SPI Tx/Rx Interrupt Handler

This method is responsible for ensuring that the data is transmitted and received correctly. It works in much the same way as the previous buffered SPI example. The main difference between this module and the previous example is what happens when the first byte of the data received is equal to 0xfe. In this case the Tx buffer pointer is moved to point to the module ID. This ensures that the Netduino Go receives the correct response to the enumeration request.

Connecting the Boards

The application code contains a number of *#if* statements to take into account the differing pin layouts of the microcontrollers used. The following have been tested so far:

- STM8S103F3 TSSOP20 on a breadboard
- STM8S Discovery

The Protomodule has also been wired up for one particular module but at the time of writing the definitions have not been added to the sources.

In order to connect the Netduino Go main board to a module in development you will probably need to purchase some form of breakout such as the Komodex breakout board (Straight connectors or 90-Degree connectors).

Connecting the two boards should be a simple case of ensuring that the SPI pins are connected MOSI to MOSI, MISO to MISO, CS to CS and Clock to Clock. In the case of the Discovery board I used PB0 for the CS line and for the STM8S103 setup I used the standard pin PA3.

Running the Code

Running the code should be a simple case of loading the STM8S code into the IAR development environment first and the deploying the code to the chip. Hot F5 to run the code.

Next, load the visual studio code and deploy this to the Netduino Go main board. Hit F5 to run the code.

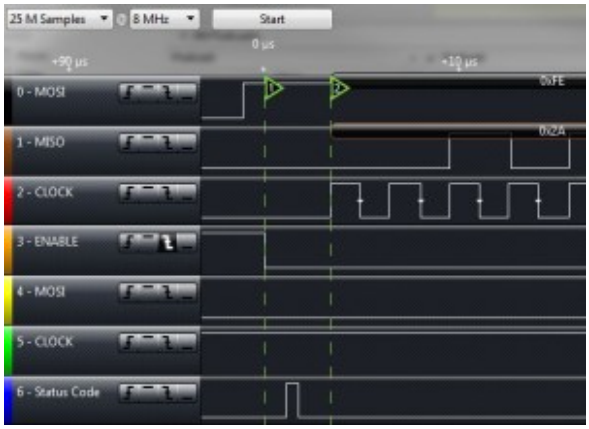
The C# code running in Visual Studio should start to print some diagnostic information to the debug window. You should see a series of statements of the form *Adding 5 to 6 to give 11*. The 6 is the parameter which has been sent to the module for processing and the 11 is the result.

result in an exception in Visual Studio. To date I have only seen this behaviour with the STM8S103 setup. The STM8S Discovery board seems to work correctly. If you have problems with this then the only suggestion is to detach IAR from the board and rely upon diagnostic information being dumped to a logic analyser. You will note that the test application which runs on the Netduino Go has the instantiation of the module wrapped in a while loop and a try block. This allows the test code to make several attempts at creating a new module instance. This should not be necessary in the final production code as this has not yet failed in a none debug environment.

This code has been tested with the simple module example here and also with a temperature and humidity sensor. The application enumerated OK and has been soak tested in two environments over a period of hours. The code seems to be stable and works well with the Netduino Go.

I originally tried to be ambitious with the interrupt service routine dealing with the chip select line. This gave me code which was simpler but lead to a timing issue. As it stands at the moment, dropping the chip select line from high to low starts the SPI processing. The time between this happening and the first clock transition is only 3.25us as measured on my logic analyser. This means that all of the preparation must be completed in 3.25us.

If we look at the diagram below you can see the timings at the start of the SPI communication:



SPI Timing Diagram

The two markers 1 & 2 indicate the time we have between the start of the comms indicated by CS falling to the first clock pulse. The *Status Code* trace is a debugging signal generated by the application. The rising edge indicates when the first line of the interrupt service routine for the CS line starts and the falling edge indicates the point where we have completed enough processing to allow SPI to be enabled.

Conclusion

This post shows how we to create a Netduino Go module using a standard communication protocol. Additional module functionality can simply be added by adding to the function table.

As noted at the start, this article is the combination of information provided by Netduino community members along with the module code which can be found in the Wiki.

As usual, the source code for this application is available for download (STM8S Go Module and Netduino Go – Basic Module Driver).

Source Code Compatibility

System	Compatible?

Tags: Electronics, Netduino, Software Development, STM8, The Way of the Register

Monday, November 26th, 2012 at 9:01 pm • Electronics, Netduino, Software Development, STM8 • RSS 2.0 feed Both comments and pings are currently closed.

Comments are closed.

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