More Serial Ports! SPI-to-Serial for .NET Micro Framework v2.2

By John R. Malin and Sean D. Liming
SJJ Embedded Micro Solutions

August 2010

Overview

RS-232 has been around since the beginning of modern computing. In 1969, a standard was introduced, and since then RS-232 can be found in almost every computing system. Even modern system-on-chips (SOC's) offer a few serial ports. With many embedded systems moving to SOC solutions, finding the right mix of features and capabilities can make it difficult to support all desired hardware IO configurations directly from the SOC.

After listening to customer feedback on different features needed in .NET Micro Framework, serial support is one of the most requested. "Need more serial ports!" Since SOCs limit the number of serial ports because of the pins available on the chip, another serial port, SPI, offers a way to add more RS-232 ports to the system. SPI is an open bus architecture that can support a number of devices such as Analog-to-Digital converters, Real-Time controllers (RTC), SPI-to-Ethernet modules, GPIO expanders, 7-segment LED drivers, various PIC controllers, etc. Best of all you can interface a number of devices from a single SPI port and make the devices address selectable. In the end you can have virtually as many devices as there are GPIO lines to enable them.

The Hardware

Of particular interest, is a SPI-to-RS-232 solution. A brief search of the Internet results in a solution from Maxim Integrated Products (http://www.maxim-ic.com/). Maxim provides a couple of IC solutions – MAX3110E (5V) and MAX3111E (3V). Both chips come in DIP packaging, which makes it easy to prototype. Figure 1 shows the circuit for the MAX3110E connected to EMAC's iPac-9302 board. This solution is for a single SPI-to-RS-232, but by adding an OR gate to OR a GPIO line and processor /CS line to control the /CS line of the MAX3110E/3111E, you can add other SPI devices to the SPI bus. For this exercise, we will keep the solution simple.

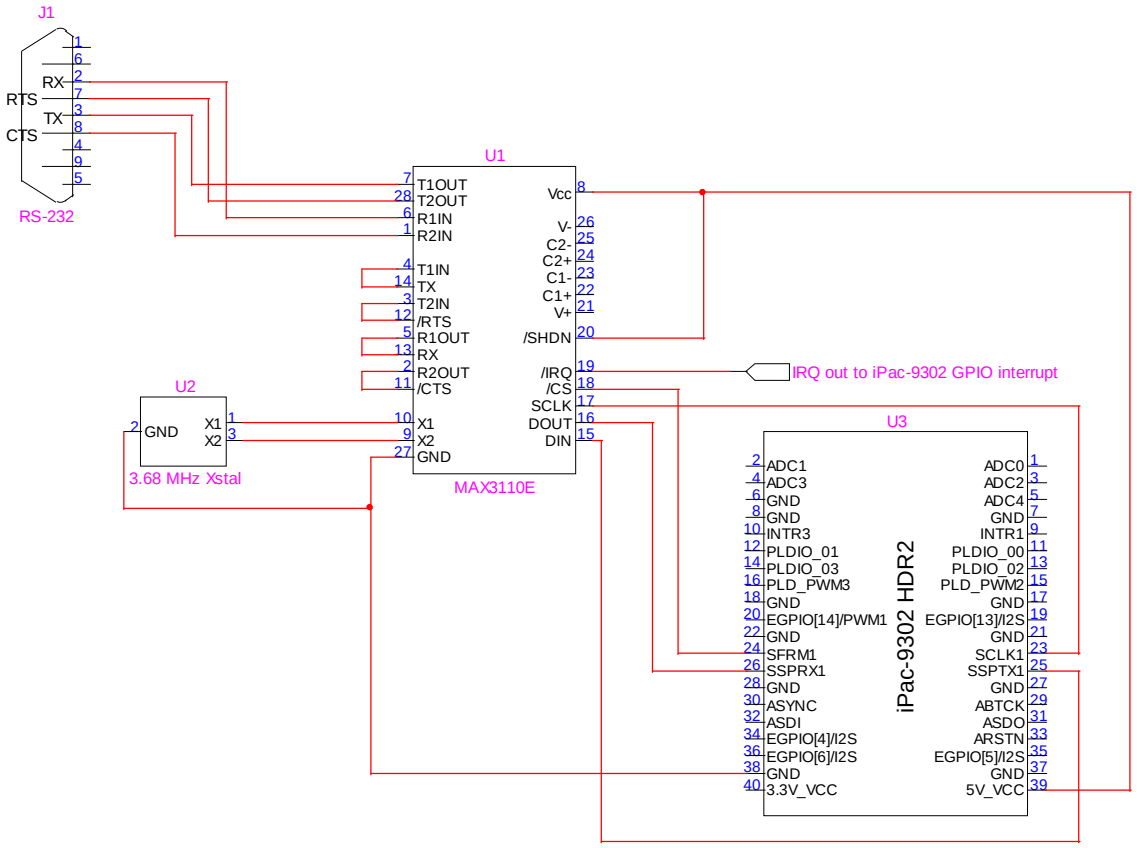

Figure 1 SPI-to-Serial Solution for the iPac-9302

Parts List:

1. MAX3110E or MAX3111E DIP or SMT – DIP makes it easier to bread board. Make sure you wire the correct supply voltage.
2. 3.68 MHz Ceramic Resonator. DigiKey part number – XC1017-ND. Manufacturing P/N: ZTT-3.68MG
3. Jumper Wire
4. 9-pin RS-232 Connector
5. Optional: OR-gate for connecting other SPI devices to the SPI bus and addressing them via GPIO line ORed with the chip select (SFRM1).

The earlier versions of .NET Micro Framework didn't support serial interrupts. This was a problem for internal SOC serial ports, because polling had to be used to capture receive data. Since V4.0 of the .NET Micro Framework, serial handling has been improved and serial interrupts are now supported. As you can see from Figure 1, the MAX3110E/3111E offers an interrupt signal that can be connected to any of the interruptible GPIO lines on the iPac-9302. Thus, not only do the Maxim devices provide a means for adding RS-232 ports to the .NET Micro Framework device, it also enhances the serial interface, by allowing them to be interrupt-driven.

The size of the data exchange supported by the iPac-9302 SPI hardware can be programmed for 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 and 16-bit data transactions. .NET Micro Framework only supports 8 or 16-bit data transactions. The MAX3110E and MAX3111E take advantage of the full 16-bit data transfers to provide control and data in each transaction. When connected to a scope you can visually see the SPI data transfers.

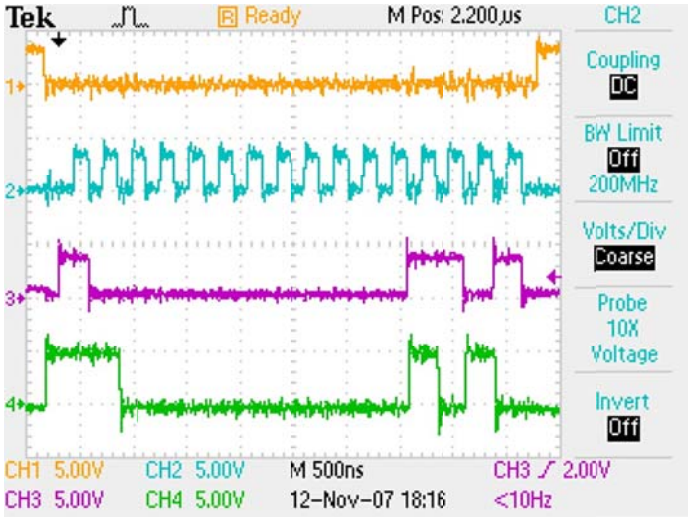

Figure 2 SPI Interface Traces of the MAX3110E

Figure 2 above shows a basic 16-bit data exchange between the iPac-9302 master-host and the MAX3110 client device. The orange signal (1) is the chip select. The aqua signal (2) is the clock. The purple signal (3) is the SSPTX1 (SPI Serial Output) data to the MAX3110. The green signal (4) is the SSPRX1 (SPI Serial Input) data, which is data sent back from the MAX33110. The MAX3110's SPI interface can run at the maximum SPI clock frequency of the iPac-9302, which is 4MHz.

The Application

Using the SPI-to-RS-232 circuit, we can write an application to communicate with application terminal emulation program, like Tera Term Pro (which can be downloaded from http://logmett.com/index.php?/download/tera-term-466.html). This simple application for the iPac-9302 receives incoming RS-232 data on interrupt and echoes it back to the sender. The source code for this example application is as follows:

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

namespace SPI_Serial
{
    public class Program
    {
        public static void Main()
        {
            Debug.Print(Resources.GetString(Resources.StringResources.String1));
            Debug.Print(Microsoft.SPOT.Hardware.SJJ.Version.HWProviderVer);
            App myApp = new App();
            myApp.Run();
        }

        public class App
        {

            static SPI.Configuration myMaximSPIPortConfig = new
SPI.Configuration(Cpu.Pin.GPIO_NONE, false, 0, 0, false, true, 5000, SPI.SPI_module.SPI1);
            SPI mySerialSPIPort = new SPI(myMaximSPIPortConfig);

//->            const UInt16 uiMaximConfig = 0xC00C;   // 4800, 8, N, 1; no interrupts
//->            const UInt16 uiMaximConfig = 0xC00B;   // 9600, 8, N, 1; no interrupts
```

```
const UInt16 uiMaximConfig = 0xC001;   // 115,200, 8, N, 1; no interrupts
const UInt16 uiMaximConfigRM = 0x0400;  // Enable receive interrupt
const UInt16 uiMaximWritePrefix = 0x8000;   // OR with 8-bit data on lower byte
const UInt16 uiMaximTxDisable = 0x0400;  //Disable char transmit on data write
const UInt16 uiMaximSetRTS = 0x0000;  //Halt incoming data: ~RTS = 0 -> RTS= 1;
                                      //update RTS only no char write
const UInt16 uiMaximClearRTS = 0x0200;  //Enable incoming data: ~RTS = 1 -> RTS= 0;
                                      //update RTS only no char write
const UInt16 uiMaximReadCommand = 0x0000;
const UInt16 uiMaximReadDataValid = 0x8000;   // AND with DOUT and > 0 if data valid
const UInt16 uiMaximTxFull = 0x4000;   // AND with DOUT and if 0 Tx full
const UInt16 uiMaximReadDataMask = 0x00FF;   // AND with data read for data out
const ushort uiMaximCTSMask = 0x0200;   // When CTS set, tranmits can be done

ushort[] uiReadCommand = new ushort[] { uiMaximReadCommand };
ushort[] uiReadData = new ushort[1];
ushort[] uiWriteData = new ushort[1];
ushort[] uiWriteReadData = new ushort[1];
ushort[] uiReadStatus = new ushort[1];
ushort uiMaximCTSStatus = 0;

InterruptPort EGPIO2;
OutputPort myGreenLED;

// Set up a line buffer
const int c_iBuffSize = 4096;
ushort[] uiLineBuffer = new ushort[c_iBuffSize];    //circular buffer
char[] cOutString = new char[c_iBuffSize];
static int iWriteIndex = 0;
static int iReadIndex = 0;



public void Run()
{


    EGPIO2 = new InterruptPort(Pins.EGPIO2_HDR3_15, false, Port.ResistorMode.Disabled,
                               InterruptModes.InterruptEdgeLow);
    EGPIO2.OnInterrupt += new NativeEventHandler(EGPIO2_OnInterrupt);

    myGreenLED = new OutputPort(Pins.GREEN_LED, false);

    // Write serial port configuration
    uiWriteData[0] = uiMaximConfig;
    uiWriteData[0] |= uiMaximConfigRM;  //Enable receive interrupt
    mySerialSPIPort.WriteRead(uiWriteData, uiReadData);

    // Enable incoming serial data, Clear RTS
    uiWriteData[0] = uiMaximWritePrefix | uiMaximTxDisable | uiMaximClearRTS;
    mySerialSPIPort.WriteRead(uiWriteData, uiReadStatus);
    uiMaximCTSStatus = (ushort) (uiMaximCTSMask & uiReadStatus[0]);

    // Character echo loop & status LED flash loop
    uint iLoopCount = 0;
    uint iValidReceiveCount = 0;
    bool bLEDState = true;
    while (true)
    {
        // Check read buffer and echo back the characters
        if ((iWriteIndex != iReadIndex) && (uiMaximCTSStatus != 0))
        {
            uiWriteData[0] = uiLineBuffer[iWriteIndex];
            uiWriteData[0] &= uiMaximReadDataMask;
            uiWriteData[0] |= uiMaximWritePrefix;
            mySerialSPIPort.WriteRead(uiWriteData, uiWriteReadData);

            if ((uiWriteReadData[0] & uiMaximTxFull) != 0)
            {
                // If write buffer not full, increment the index
                // otherwise send it next time around
```

```
                    iWriteIndex++;
                    if (iWriteIndex >= c_iBuffSize)
                    {
                        // Wrap index
                        iWriteIndex = 0;
                    }
                }

                // Store any valid data read during the write
                if ((uiMaximReadDataValid & uiWriteReadData[0]) != 0)
                {
                    uiLineBuffer[iReadIndex++] = uiWriteReadData[0];

                    // Check for read index wrap
                    if (iReadIndex >= c_iBuffSize)
                    {
                        // Wrap buffer
                        iReadIndex = 0;
                    }
                }

                // Enable incoming serial data, Clear RTS; check CTS
                uiWriteData[0] = uiMaximWritePrefix | uiMaximTxDisable | uiMaximClearRTS;
                mySerialSPIPort.WriteRead(uiWriteData, uiReadStatus);
                uiMaximCTSStatus = (ushort)(uiMaximCTSMask & uiReadStatus[0]);

            }

            //Idle loop; flash green LED to show activity without using a Thread.Sleep
            if ((iLoopCount++ % 1000) == 0)
            {
                if (bLEDState)
                {
                    bLEDState = false;
                }
                else
                {
                    bLEDState = true;

                    //Output the valid receive char count on WriteRead
                    if (iValidReceiveCount > 0)
                    {
                        Debug.Print(iValidReceiveCount.ToString());

                        // Reset count
                        iValidReceiveCount = 0;
                    }
                }
                myGreenLED.Write(bLEDState);
            }
        }



    }

    void EGPIO2_OnInterrupt(uint data1, uint data2, DateTime time)
    {
        // Do NULL reads only in this interrupt handler; keep it lean & mean
        mySerialSPIPort.WriteRead(uiReadCommand, uiReadData);

        // Check CTS
        uiMaximCTSStatus = (ushort)(uiMaximCTSMask & uiReadData[0]);

        //If data valid, store it in the circular buffer; keep it lean & mean
        if ((uiMaximReadDataValid & uiReadData[0]) != 0)
        {
            uiLineBuffer[iReadIndex++] = uiReadData[0];

            // Check for read index wrap
            if (iReadIndex >= c_iBuffSize)
```

```
                    {
                        // Wrap buffer
                        iReadIndex = 0;
                    }
                }
            }
        }
    }
}
```

The Main routine simply calls the Run method, which is the main body of the application. The first part of the Run method sets up the iPac-9302's FGPIO as an interrupt port, the green LED for output, and configures the MAX3110E serial communications. You will see 3 configuration statements that illustrate how to set the MAX3110E for different baud rates. The examples for 4800, 8, N, 1 and 9600, 8, N, 1 are commented, and we will use 115200, 8, N, 1for this example.

The second part of the Run method is the main program loop. It is an infinite loop that monitors a circular buffer for characters to transmit, transmits any characters it finds in the circular buffer out the MAX3110E serial port, manages RTS/CTS flow control, and toggles the green LED as a status indicator that the program is running.

The final part of this program is the interrupt handler. It is called when incoming interrupts from the MAX3110 indicate that there is data coming in. The interrupt handler reads the data and checks it to see if it is valid. Then the received data is queued for transmit to the circular buffer that the main program loop is monitoring, so it can be echoed back out the serial port.

Simply echoing RS-232 data is a pretty basic test program, but there are important features of this test program that should be noted. By storing the received data in the circular buffer and letting the main program loop do the data transmission, or echo, this keeps down the processing overhead in the interrupt handler, keeping it lean and mean. This allows the interrupt handler to handle the receive interrupt and be ready for the next character as quickly as possible. The transmission of the echoed characters is managed by the main program loop and can be interrupted by incoming characters.

The desire is to not lose any incoming characters, but with a non-real-time system, like .NET Micro Framework, this can be a challenge. Keeping interrupt latency to a minimum, managing the data stream flow-control, and controlling character burst rate are strategies that can be employed, but there are certain timing aspects of the SPI-Serial chip in conjunction with the internal design of the .NET Micro Framework that cannot be changed. The .NET Micro Framework uses an ISR/IST interrupt management model. That is, there is an interrupt service routine sitting on the hardware fielding the hardware interrupts as they come in. The ISR then determines whom the interrupt is for and hands it off to the appropriate interrupt service thread. This IST then calls the registered interrupt handler that is in the managed code of your application. There is latency associated with this process. The .NET Micro Framework scheduler determines what threads will be run in what priority, and the .NET Micro Framework scheduler is not preemptive. That is to say that when the ISR signals the scheduler to activate a particular IST, the IST is given priority over normal application process threads, but it may not be immediately scheduled to run. If there is another thread running, even at a lower priority than the IST, the scheduler will not preempt that thread and activate the IST. If all application process threads are suspended, then the IST will run immediately, otherwise the IST will be queued to run the next time the scheduler reschedules threads. This will happen if the active thread does a thread sleep or is suspended for any reason, or it will eventually happen on the 20 mSec time hack that the scheduler responds to for mandatory thread scheduling. Therefore, the response to an interrupt can be as quick as the ISR/IST response latency or it can be as long as 20 mSec plus the ISR/IST response latency. If you have repeated interrupts coming faster than 20 mSec, there will likely be times when an interrupt is missed.

When using this application to echo characters that are entered from a keyboard as someone types, there is little likelihood of missing an interrupt and dropping a character. An average typist can sustain a typing rate of about 60 words-per-minute, and a typical fast typist can sustain bursts of characters at 80 words-per-minute. These rates are standardized for an average word size of 5 characters-per-word. Therefore, a fast typist can send bursts of characters from a keyboard at 400 characters-per-minute or 150 mSec per character. This is more than 7 times the worst-case interrupt response time, so one would not expect to miss any interrupts from human generated data streams. If we were to burst characters over the RS-232 link by sending a file, we could send the characters with virtually no transmit delay between characters. We have configured the RS-232 link to operate at 115200 baud (bits-per-second), with an 8-bit character and a stop bit. Each time an RS-232 character is sent out there is an initial start-bit, followed by the character bits, and finally the stop-bit. Therefore, there are 10 bits-per character. At 115200 baud, each bit is almost 9 uSec long. Each character would then be almost 90 uSec long. This is significantly shorter than the worst-case interrupt latency. The MAX3110E helps us by having an 8-character internal buffer. At 115200 baud, it would then take nearly 720 uSec or .72 mSec to fill the MAX3110E's receive buffer. This is still significantly shorter than the worst-case interrupt latency, so characters could still be dropped periodically.

The MAX3110E provides hardware flow-control with the RTS/CTS signals. When a character is transmitted out the RS-232 link, the MAX3110E sets RTS. If the sending device is configured for RTS/CTS, hardware flow-control, it will stop sending until RTS is cleared. This application manages that by leaving RTS set until the main program loop has completed sending a character from the circular buffer and has completed managing the circular buffer pointers. During that time the sending device will not send characters. This helps to throttle the data stream to keep it from overrunning the receive buffer, but managing RTS is done through the SPI interface as well. The writing of a SPI command must be managed by the scheduler as well and therein is the rub. You cannot throttle the data stream fast enough to manage all timing conditions when you have to use the SPI interface as a control interface, as well as, a data read and write interface. This means that we are left with introducing delays between characters to keep the data stream at a manageable rate for the .NET Micro Framework device. It should be obvious that introducing a 20 mSec delay between each character in burst mode will guarantee that no characters will be dropped when the interrupt latency is at its maximum, but with RTS flow control, we can actually reduce the inter-character delay to 17 mSec and have bursts of characters received without dropping any characters. This particular application was tested with bursts of 240 characters and 17 mSec inter-character delay, and all characters were received and echoed repeatedly. Figure 3 shows the Tera Term serial port configuration parameters that were used to test the application, note the transmit delay of 17 mSec/char and hardware flow-control enabled.

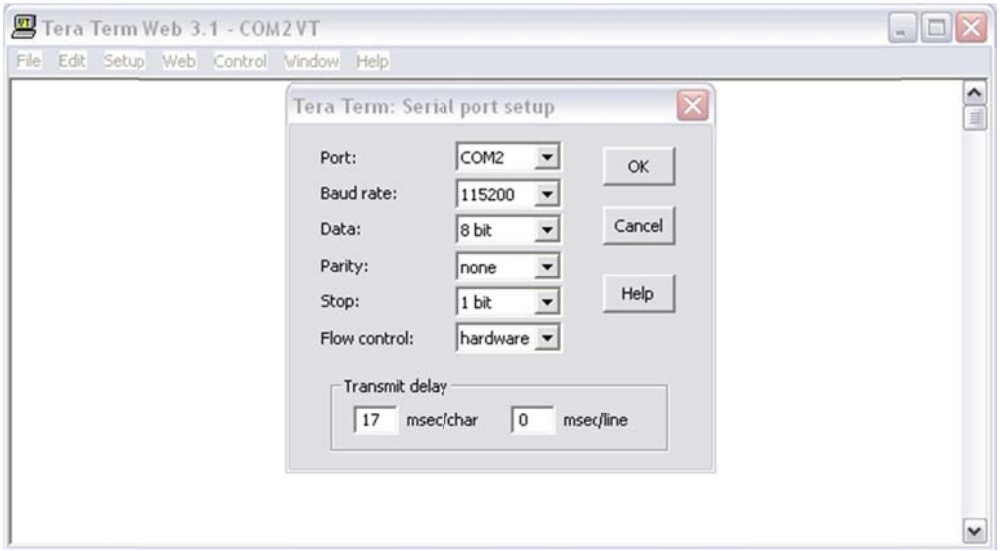

Figure 3 Tera Term Serial Port Configuration

Finally, to manage a slow sending device, CTS must be monitored; and no data must be transmitted when CTS is set, which is controlled by the RTS line of the sending device. At the top of the main program while-loop, CTS is tested and characters will not be transmitted if the sending device has set its RTS. CTS is checked whenever RTS is cleared in the main program loop and in the interrupt handler when each character is received. This completes the RTS/CTS flow-control model and keeps the application from overrunning the sending device when echoing characters.

Managed Code Driver

To simplify the application code, some of the more mundane operations that must be done through the SPI to configure the SPI-Serial device, control the SPI-Serial device, and send and receive data, as well as, the bit fields and other constants can be wrapped into a managed code driver class. The following is the source code listing of an example managed code driver, SPI_Serial_Driver:

```
using System;
using Microsoft.SPOT;
using Microsoft.SPOT.Hardware;
using System.IO.Ports;

namespace Microsoft.SPOT.Hardware.SJJ.SPI_Serial_Driver
{
    public static class Version
    {
        public const string SpiSerVersion = "SJJ SPI-Serial Driver V4.1.1.1";
    }

    public class SPI_Serial_Driver
    {
        //Default configurations
        protected BaudRate ssBaudRate = BaudRate.Baudrate115200;
        protected Parity ssParity = Parity.None;
        protected int ssDataBits = 8;
        protected StopBits ssStopBits = StopBits.One;
        protected ushort[] usSpiSerConfig = new ushort[1];
        protected ushort[] usSpiSerRTS = new ushort[1];
        protected ushort[] usWriteChar = new ushort[1];

        private const UInt16 uiMaximConfigRM = 0x0400;  // Enable receive interrupt
        private const UInt16 uiMaximWritePrefix = 0x8000;    // OR with 8-bit data on lower byte
        private const UInt16 uiMaximTxDisable = 0x0400;  //Disable char transmit on data write
        private const UInt16 uiMaximSetRTS = 0x0000;  //Halt incoming data: ~RTS = 0 -> RTS= 1; update RTS
                                                      // only no char write
        private const UInt16 uiMaximClearRTS = 0x0200;  //Enable incoming data: ~RTS = 1 -> RTS= 0;
                                                        // update RTS only no char write
        private const ushort uiMaximCTSMask = 0x0200;    // When CTS set, tranmits can be done
        private const UInt16 uiMaximReadDataMask = 0x00FF;    // AND with data read for data out
        private const UInt16 uiMaximReadCommand = 0x0000;
        private const UInt16 uiMaximReadDataValid = 0x8000;    // AND with DOUT and > 0 if data valid
        private const UInt16 uiMaximTxFull = 0x4000;   // AND with DOUT and if 0 Tx full

        public ushort[] uiReadCommand = new ushort[] { uiMaximReadCommand };


        //Constructor
        public SPI_Serial_Driver()
        {
            usSpiSerConfig[0] = 0xC001; //115200, 8, N, 1
        }

        //Overloaded Configuration method
        public ushort[] SPI_Serial_Config()
        {
            //Enable receive interrupt
```

```
            usSpiSerConfig[0] |= uiMaximConfigRM;

            // Return constructor default configuration
            return (usSpiSerConfig);
        }

        public ushort[] SPI_Serial_Config(BaudRate setBaudRate)
        {
            SetBaudConfig(setBaudRate);

            //Enable receive interrupt
            usSpiSerConfig[0] |= uiMaximConfigRM;

            return (usSpiSerConfig);
        }


        public ushort[] SPI_Serial_Config(BaudRate setBaudRate, int setDataBits)
        {
            SetBaudConfig(setBaudRate);

            SetDataConfig(setDataBits);

            //Enable receive interrupt
            usSpiSerConfig[0] |= uiMaximConfigRM;

            return (usSpiSerConfig);
        }

        public ushort[] SPI_Serial_Config(BaudRate setBaudRate, int setDataBits,
                                          StopBits setStopBits)
        {
            SetBaudConfig(setBaudRate);

            SetDataConfig(setDataBits);

            switch (setStopBits)
            {
                case StopBits.One:
                    {
                        usSpiSerConfig[0] &= 0xFFBF;
                        break;
                    }
                case StopBits.Two:
                    {
                        usSpiSerConfig[0] |= 0x0040;
                        break;
                    }
                default:
                    {
                        //default to 1 stop bit
                        usSpiSerConfig[0] &= 0xFFBF;
                        break;
                    }
            }

            //Enable receive interrupt
            usSpiSerConfig[0] |= uiMaximConfigRM;

            return (usSpiSerConfig);
        }

        public ushort[] ClearRTS()
        {
            usSpiSerRTS[0] = uiMaximWritePrefix | uiMaximTxDisable | uiMaximClearRTS;

            return (usSpiSerRTS);
        }

        public bool CheckCTS(ushort uiReadData)
        {
            bool bReturnVal = false;
```

```
            if ((uiReadData & uiMaximCTSMask) != 0)
            {
                bReturnVal = true;
            }

            return (bReturnVal);
        }

        public ushort[] WriteChar(ushort usChar)
        {
            usWriteChar[0] = (ushort) (usChar & uiMaximReadDataMask);
            usWriteChar[0] |= uiMaximWritePrefix;

            return (usWriteChar);
        }

        public bool CheckDataValid(ushort uiReadData)
        {
            bool bReturnVal = false;

            if ((uiMaximReadDataValid & uiReadData) != 0)
            {
                bReturnVal = true;
            }

            return (bReturnVal);
        }

        public bool CheckTxEmpty(ushort uiReadData)
        {
            bool bReturnVal = false;

            if ((uiReadData & uiMaximTxFull) != 0)
            {
                bReturnVal = true;
            }

            return (bReturnVal);
        }

        private void SetBaudConfig(BaudRate setBaudRate)
        {
            // Clear baudrate bits
            usSpiSerConfig[0] &= 0xFFF0;

            switch (setBaudRate)
            {
                case BaudRate.Baudrate115200:
                    {
                        usSpiSerConfig[0] |= 0x0001;
                        break;
                    }
                case BaudRate.Baudrate57600:
                    {
                        usSpiSerConfig[0] |= 0x0002;
                        break;
                    }
                case BaudRate.Baudrate38400:
                    {
                        usSpiSerConfig[0] |= 0x0009;
                        break;
                    }
                case BaudRate.Baudrate19200:
                    {
                        usSpiSerConfig[0] |= 0x000A;
                        break;
                    }
                case BaudRate.Baudrate9600:
                    {
                        usSpiSerConfig[0] |= 0x000B;
                        break;
                    }
                case BaudRate.Baudrate4800:
```

```
                    {
                        usSpiSerConfig[0] |= 0x000C;
                        break;
                    }
                case BaudRate.Baudrate2400:
                    {
                        usSpiSerConfig[0] |= 0x000D;
                        break;
                    }
                case BaudRate.Baudrate1200:
                    {
                        usSpiSerConfig[0] |= 0x000E;
                        break;
                    }
                case BaudRate.Baudrate600:
                    {
                        usSpiSerConfig[0] |= 0x000F;
                        break;
                    }
                default:
                    {
                        //default to 115200 baud
                        usSpiSerConfig[0] |= 0x0001;
                        break;
                    }
            }
        }

        private void SetDataConfig(int setDataBits)
        {
            if (setDataBits == 8)
            {
                usSpiSerConfig[0] &= 0xFFEF;
            }
            else if (setDataBits == 7)
            {
                usSpiSerConfig[0] |= 0x0010;
            }
            else
            {
                //default to 8 data bits
                usSpiSerConfig[0] &= 0xFFEF;
            }
        }
    }

}
```

Note that all the MAXIM SPI specific constants and bit fields are contained in the driver, and do not have be defined in the application. They will be hidden from the application program, and therefore, simplify the application code.

When the driver class is instantiated, the constructor initializes the default serial port settings to 115200 baud, 8 character bits, no parity, and 1 stop bit. The driver class then provides a series of methods to interface to the driver. The methods that are used in cooperation with the SPI.WriteRead() API, accept various arguments and data and return a one-element ushort buffer pointer to a properly formatted SPI write command. The application developer doesn't need to know anything about the bit-field formats for the MAXIM SPI commands.

The methods that are used with the SPI.WriteRead() are:

- SPI_Serial_Config – which is an overloaded method that allows selection of the class default configurations settings, or one can set just the baud rate, the baud rate and the data bits, or the baud rate, data bits, and stop bits using the supplied overloads.

- ClearRTS – which clears RTS to allow the sending application to send data. This command clears RTS without reading or writing any RS-232 data.

- WriteChar – causes a single character to be transmitted out the RS-232 port.

The methods that are used to analyze data that is returned, read, from the SPI device are:

- CheckCTS – checks of CTS is set and returns true or false.
- CheckDataValid – checks data read from the RS-232 port and returned through the SPI to see if it is valid received data or status only. It return true or false.
- CheckTxEmpty – check returned data status for the status of the MAXIM transmit buffer to see if there is room for the next character to be transmitted out the RS-232 port. It returns true or false.

Finally, there is a public, one-member, ushort array that contains the MAXIM read data command, that is used in conjunction with the SPI.WriteRead API to read incoming RS-232 data, and a public Version class with a SpiSerVersion string that can be accessed to verify the current version of the driver.

Note: the MAXIM chip does not support all possible RS-232 configurations, i.e. baud rates, character sizes, and stop bits. If a configuration parameter is supplied that is not supported by the MAXIM chip, the driver will use the default setting, instead.

Now that we have the MAXIM SPI interface wrapped in a managed code driver, let's look at the code listing for SPI_Serial2, with is a rewrite of the original SPI_Serial test application using the new managed code driver:

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

namespace SPI_Serial2
{
    public class Program
    {
        public static void Main()
        {
            Debug.Print(Resources.GetString(Resources.StringResources.String1));
            Debug.Print(Microsoft.SPOT.Hardware.SJJ.Version.HWProviderVer);
            Debug.Print(Microsoft.SPOT.Hardware.SJJ.SPI_Serial_Driver.
                        Version.SpiSerVersion);
            App myApp = new App();
            myApp.Run();

        }

        public class App
        {
            static SPI.Configuration myMaximSPIPortConfig = new
SPI.Configuration(Cpu.Pin.GPIO_NONE, false, 0, 0, false, true, 5000,
SPI.SPI_module.SPI1);
            SPI mySerialSPIPort = new SPI(myMaximSPIPortConfig);

            static SPI_Serial_Driver mySpiSerial = new SPI_Serial_Driver();

            InterruptPort EGPIO2;
            OutputPort myGreenLED;

            ushort[] uiReadData = new ushort[1];
            ushort[] uiWriteData = new ushort[1];
            ushort[] uiReadStatus = new ushort[1];
```

```
            ushort[] uiWriteReadData = new ushort[1];
            bool bMaximCTSStatus = false;

            const int c_iBuffSize = 4096;
            ushort[] uiLineBuffer = new ushort[c_iBuffSize];    //circular buffer
            static int iWriteIndex = 0;
            static int iReadIndex = 0;

            public void Run()
            {
                EGPIO2 = new InterruptPort(Pins.EGPIO2_HDR3_15, false,
                         Port.ResistorMode.Disabled, InterruptModes.InterruptEdgeLow);
                EGPIO2.OnInterrupt += new NativeEventHandler(EGPIO2_OnInterrupt);

                myGreenLED = new OutputPort(Pins.GREEN_LED, false);

                // Write serial port configuration

mySerialSPIPort.WriteRead(mySpiSerial.SPI_Serial_Config(System.IO.Ports.BaudRate.Baudrate
115200, 8, System.IO.Ports.StopBits.One), uiReadData);

                // Enable incoming serial data, Clear RTS
                mySerialSPIPort.WriteRead(mySpiSerial.ClearRTS(), uiReadStatus);

                // Check CTS status
                bMaximCTSStatus = mySpiSerial.CheckCTS(uiReadStatus[0]);

                // Character echo loop & status LED flash loop
                uint iLoopCount = 0;
                uint iValidReceiveCount = 0;
                bool bLEDState = true;
                while (true)
                {
                    // Check read buffer and echo back the characters
                    if ((iWriteIndex != iReadIndex) && bMaximCTSStatus)
                    {
              mySerialSPIPort.WriteRead(mySpiSerial.WriteChar(uiLineBuffer[iWriteIndex]),
                                        uiWriteReadData);

                        if (mySpiSerial.CheckTxEmpty(uiWriteReadData[0]))
                        {
                            // If write buffer not full, increment the index otherwise
                            // send it again next time around
                            iWriteIndex++;
                            if (iWriteIndex >= c_iBuffSize)
                            {
                                // Wrap index
                                iWriteIndex = 0;
                            }
                        }

                        // Store any valid data read during the write
                        if (mySpiSerial.CheckDataValid(uiWriteReadData[0]))
                        {
                            uiLineBuffer[iReadIndex++] = uiWriteReadData[0];

                            // Check for read index wrap
                            if (iReadIndex >= c_iBuffSize)
                            {
                                // Wrap buffer
                                iReadIndex = 0;
                            }
                        }

                        // Enable incoming serial data, Clear RTS; check CTS
                        mySerialSPIPort.WriteRead(mySpiSerial.ClearRTS(), uiReadStatus);
                        bMaximCTSStatus = mySpiSerial.CheckCTS(uiReadStatus[0]);
                    }

                    // Idle loop; flash green LED to show activity without using a
                    // Thread.Sleep
                    if ((iLoopCount++ % 1000) == 0)
                    {
```

```
                        if (bLEDState)
                        {
                            bLEDState = false;
                        }
                        else
                        {
                            bLEDState = true;

                            //Output the valid receive char count on WriteRead
                            if (iValidReceiveCount > 0)
                            {
                                Debug.Print(iValidReceiveCount.ToString());

                                // Reset count
                                iValidReceiveCount = 0;
                            }
                        }
                        myGreenLED.Write(bLEDState);
                    }
                }

            }

            void EGPIO2_OnInterrupt(uint data1, uint data2, DateTime time)
            {
                // Do NULL reads only in this interrupt handler; keep it lean & mean
                mySerialSPIPort.WriteRead(mySpiSerial.uiReadCommand, uiReadData);

                // Check CTS
                bMaximCTSStatus = mySpiSerial.CheckCTS(uiReadData[0]);

                //If data valid, store it in the circular buffer; keep it lean & mean
                if (mySpiSerial.CheckDataValid(uiReadData[0]))
                {
                    uiLineBuffer[iReadIndex++] = uiReadData[0];

                    // Check for read index wrap
                    if (iReadIndex >= c_iBuffSize)
                    {
                        // Wrap buffer
                        iReadIndex = 0;
                    }
                }
            }
        }
    }
}
```

What cannot be seen in the listing is that to add the SPI-Serial driver to the application a new reference must be added. From the *Project* dropdown in Visual Studio, select *Add reference…*, select the *Browse* tab, and navigate to and select SPI_Serial_Driver.dll, the SPI-Serial driver DLL. Note that there is a new using statement to reference the namespace of the SPI-Serial driver: `using Microsoft.SPOT.Hardware.SJJ.SPI_Serial_Driver;`

Note that all the MAXIM SPI-related constants and bit fields have been removed. Next, check each of the mySerialSPIPort.WriteRead() statements. You can see that each now uses either a driver method or a driver public variable to provide the appropriately formatting SPI command. Finally, you will see that each of the status check, for CTS, valid read data, or TX buffer status have been replaced with driver methods that can be simply tested for true or false.

Note that there would be 2 ways we could have dealt with putting the circular buffer into the driver: simply making it a public variable that you could access just the same way you access the one currently in the application code or putting a class method wrapper around it. There is a lot of call overhead when calling a class method that is way more than calling a simple function. This method would have had to be called in the interrupt handler. We already have the read class method in the interrupt handler, so in an effort not to burden the interrupt handler with another class method call, the best the way it is to leave it to the application developer to

implement the circular buffer. This improves interrupt handler performance and gives the application developer complete flexibility on the implementation of this buffer.

Clearly, for this application or any other SPI-Serial applications using the MAXIM SPI-Serial chip, the SPI-Serial driver simplifies the code structure and allows the software developer to concentrate on the specifics of the application and not the complexities of the MAXIM SPI-Serial chip.

Windows is a registered trademark of Microsoft Corporation.
MAXIM is a registered trademark of Maxim Integrated Products.