2010

Embedded Development Kit for the Microsoft® .NET Micro Framework

Basic Step-by-Step Guide for .NET Micro Framework Application Development on the iPac-9302

By Sean Liming & John R. Malin
SJJ Embedded Micro Solutions, LLC

1/8/2010

First Release: September 2007
Second Release: January 2008
Third Release: June 2008
Fourth Release October 2008
Fifth Release January 2010

Published in the United States by

SJJ Embedded Micro Solutions, LLC.
6432 Glendale Dr.
Yorba Linda, CA 92886 USA

www.sjjmicro.com

Safety Information

Warning: To prevent fire or shock hazard, do not expose product to water, rain, or any type of moisture.

Always follow basic safety precautions when using this product to reduce risk of injury from fire or electrical shock

Warning Potential shock hazard

1. Read and understand all safety instructions.
2. Make sure power is disconnected before making any circuit connections.
3. Use a ground strap when work with hardware.
4. Place power cord so that no one can step or trip on it.
5. Do not use this product near water or an moisture source.

Table of Contents

1 Welcome

Welcome to the Microsoft® .NET Micro Framework, one of the most exciting embedded technologies in some time. The .NET Micro Framework or .NET MF lets developers focus on the application part of an embedded system while removing the tedious task of developing low level device drivers. Using the familiar Microsoft® Visual Studio development suite, anyone from engineers and software developers to students can write applications that directly control hardware devices.

The .NET Micro Framework consists of the .NET Framework Common Language Runtime (CLR) sitting on top of a hardware abstraction layer (HAL) that allows developers to write managed code applications in Visual Studio for small footprint devices. Using only a few hundred kilobytes of RAM and an inexpensive processor, the Microsoft .NET Micro Framework platform gives you the means to build applications by using the same development tools and advanced languages, such as C#, that you use to build desktop applications.

The Embedded Development Kit for .NET Micro Framework (EDK) provides the foundation for writing C# applications for the .NET MF. This guide provides the background and basic step-by-step instruction to get started.

1.1 EDK Background

Reusable code is one of the Holy Grails for software developers. In this effort, abstracting the hardware from the OS details and focusing on the application became an integral part of this quest. In the late 1970's, Object Oriented Programming (OOP) was a means to address this concept, and the evolution of different solutions from MFC, Java, and now .NET were a means for practical implementation. For application development, these solutions have worked well, and progress is being made in new programming concepts and development tools.

Because of the abstraction from the hardware, the idea of reusable code breaks down when it comes to interfacing to the hardware. Embedded developers are used to writing code that interacts directly with the hardware. Every embedded controller manufacturer implements their internal chip architecture differently. Even if the CPU core is the same, the surrounding busses, interrupt controllers, memory interfaces, and IO are all different. Writing software for one CPU family and trying to run it to a different CPU family will not work. Code has to be re-written natively and tested.

The hardware abstraction layer (HAL) was developed to help operating systems separate the kernel from the hardware, thus making the kernel portable from platform to platform. Windows NT, 2000, XP, and Vista all make us of the HAL. Writing the HAL / device drivers take time and experience.

Developing an embedded device from the ground up has always been a challenging task and not for the faint of heart. As you would expect, board bring-up is the critical path for any project. .NET MF integrated to a standard hardware platform provides a different approach. The code to interact with the hardware is already complete; all you have to do is write the managed application. The EDK has been design to help eliminate this critical path so you can concentrate on your application.

The EDK features EMAC's iPac-9302 Single Board Computer (SBC) with the .NET Micro Framework already on board. The iPac-9302 features an ARM9 processor with a variety of IO to meet the needs of any application. The EDK is a multi-use solution that can be used to develop a variety of embedded applications such as robotics, shipment tracking, industrial controls, security systems, Point of Service applications, and much more. The iPac-9302 is a production-quality board that can be embedded, as-is, into a final product; or it can be used as a reference design for a custom end product.

1.2 Basic introduction to the .NET Micro Framework

Windows XP and Windows 7 support the full .NET Framework. Windows CE supports .NET Compact Framework which is a subset of the full .NET Framework.

OS	Framework
Windows XP / XP Embedded / Vista	.NET Framework 1.1, 2.0, 3.0, 3.5
Windows CE	.NET Compact Framework
.NET Framework CLR + HAL	.NET Micro Framework

Table 1.1 - OS and .NET Support

.NET Micro Framework is an even smaller subset, and only contains support for what is relevant for small footprint devices. Designed from the ground up, .NET MF will run battery powered devices with lower-end 32-bit processors with minimal flash and RAM. The SPOT watch and the Windows Vista SideShow display are examples of earlier implementations of the technology. Beyond these applications, .NET MF serves as an ideal platform for deeply embedded applications where Microsoft's other embedded offerings, Windows CE and Windows XP Embedded, are too large to fit.

.NET MF can be ported to different ARM-based processors, which has already been completed for the iPac-9302 / EDK. The cost of many embedded projects goes up because integrating the software to the hardware can be complex. The advantage of the EDK is that the .NET MF has already been ported to the iPac-9302 hardware and is ready to run applications developed in Visual Studio. The EDK is a generic development platform that allows you to download different applications to control different devices.

1.2.1 Intro to the CLR

As stated earlier, the .NET MF consists of a CLR. CLR means Common Language Runtime. The CLR is an agent that manages code at execution time, providing core services such as memory management, thread management, exception handling and debug services, while also enforcing strict type safety and other forms of code accuracy that promote application security and robustness. In fact, the concept of code management is a fundamental principle of the CLR. Code that targets the CLR is referred to as managed code, whereas code that does not target the CLR is known as unmanaged code.

Programmers writing in VB.NET, Visual C++ /CLR, or C# compile their programs into an intermediate form of byte code called Common Intermediate Language (CIL) in a portable execution (PE) file. In the case of .NET Framework, the code is compiled in Microsoft Intermediate language, which is Microsoft's implementation of the CIL. We will talk about what gets executed by the CLR a little later.

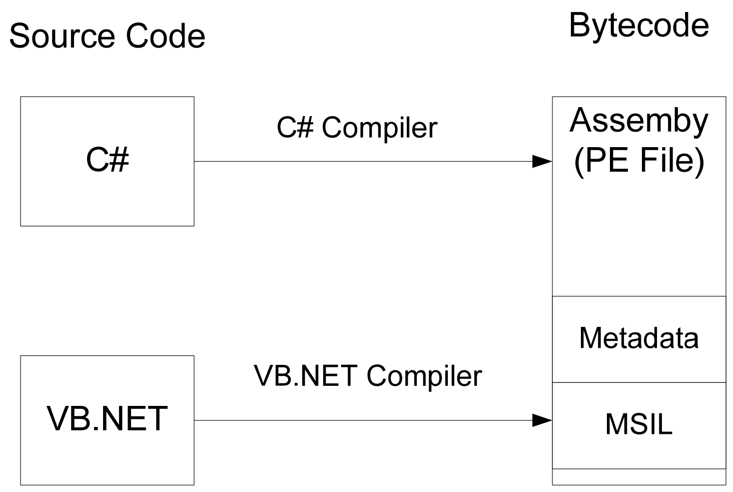


Fig 1.1 - Languages and the Bytecode

The CLR is a special implementation of the full CLR used in .NET Framework. For .NET MF, the CLR sits on top of a HAL layer, which makes it possible to port .NET MF to other processors. Think of .NET MF as a CLR (managed code) kernel, and not a full blown operating system. It is very important to know that .NET MF is not a real-time kernel. There are no deterministic responses that can be made from the CLR.

If you are familiar with C#, you will see much of the same coding techniques. If you are new to C# programming, the EDK provides a fun way to learn while implementing applications on real hardware other than a PC. A C# primer is supplied in the appendix.

1.2.2 .NET MF Architecture

.NET MF consists of 4 layers: Application, Class Library, Runtime Component, and Hardware.

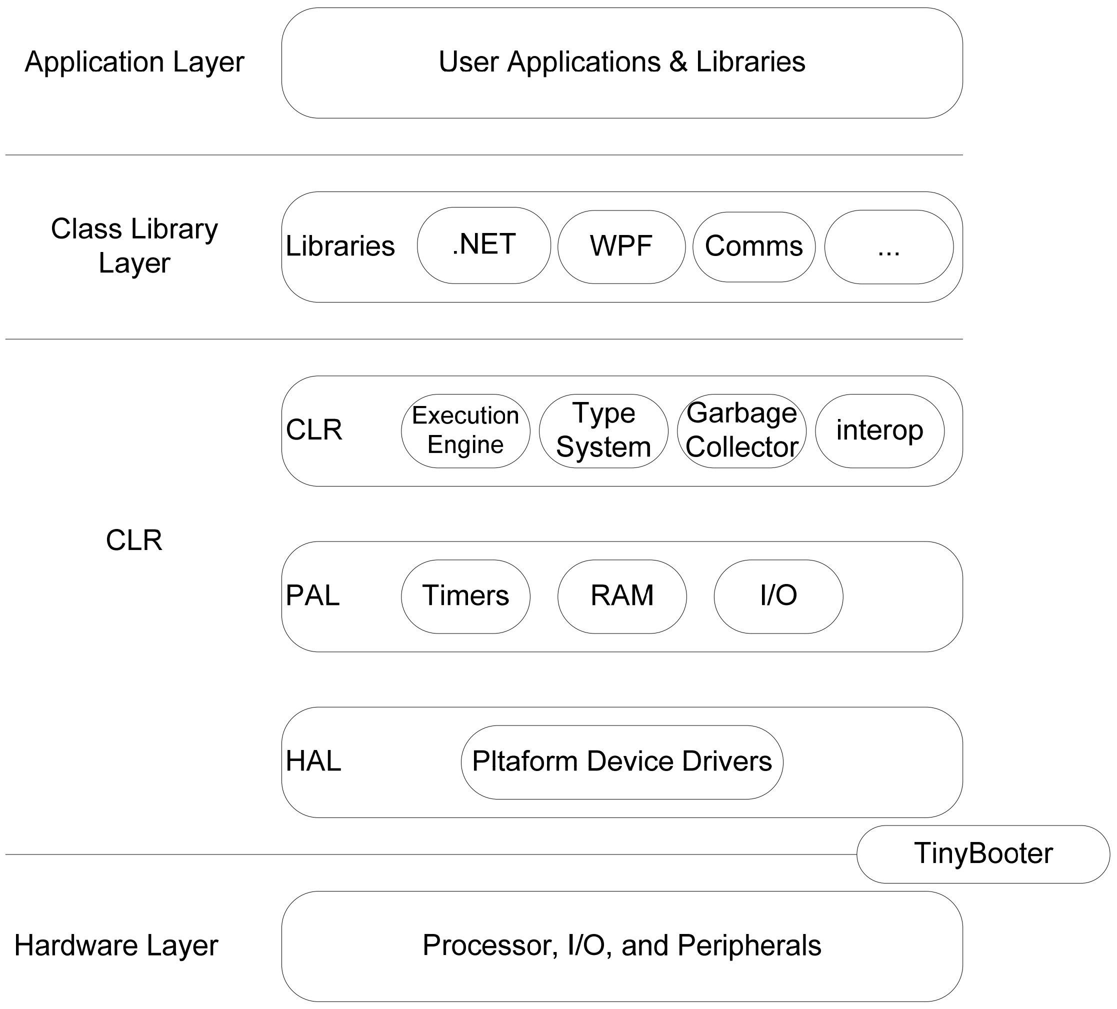


Fig 1.2 - 4 Layers of .NET MF

Application Layer – The top layer simply consists of the managed applications you write in Visual Studio. These applications are specific to the hardware that you are writing them too. Even though Visual Studio and .NET supports different languages, C# is the only language currently available to write .NET MF managed code applications

Class Library Layer – Earlier, we talked about reusable code. The class libraries are the reusable objects that you can use to write applications. The different objects that are support are

- Extended Chipset to access specific peripherals such as GPIO, serial, SPI, I2C, etc
- The CLR API class library and the CLR corelib
- Encryption
- Debug, graphics, and shell DLLs

CLR – This layer consists of three components: the .NET Micro Framework common language runtime (CLR), a hardware abstraction layer (HAL), and a platform abstraction layer (PAL).

As we described earlier, the CLR is a subset of the .NET Framework CLR, which is the run-time environment provided by the .NET Framework.

Features of the .NET Micro Framework CLR include:

- Numeric types, class types, value types, arrays (one-dimensional only), delegates, events, references, and weak references
- Synchronization, threads, and timers
- Reflection
- Serialization
- Garbage collection
- Exception handling
- Time classes, including DateTime and TimeSpan (using INT64 arithmetic)
- Time-sliced thread management
- Exceptions to and extensions of the CLR include:
    - Execution constraints that limit call durations and prevent failures.
    - Strings represented internally as UTF-8 and exposed as Unicode.
    - A global, shared string table for well-known values (types, methods, and fields) that reduces RAM and ROM and reduces the number of cross-references among assemblies.
    - A WeakDelegate class to handle dangling references to delegates.
    - Support for extending the hardware peripherals such as GPIO, serial, SPI, or I2C so that you can access them in C#.

Note*: The release CLR code is controlled by Microsoft, but starting with V4.0, the release is an Open Source release. The CLR code can now be customized and enhanced by anyone producing a CLR port. If these enhancements are submitted back to the Microsoft .NET MF Team and approved, they will be included in future Porting Kit releases and supported by the Microsoft .NET MF Team and the Open Source Porting Community. If the customizations and enhancements are kept proprietary, then support must be provided by the team or company that produced the custom port. The EDK contains HAL driver customizations and extensions for the iPac-9302 hardware. It does not contain any customizations or enhancements to the CLR/PAL code.*

Features not supported:

- Multidimensional arrays.
- Virtual tables for method resolution, which saves RAM.
- Preemptive call backs.

The HAL and the PAL control the underlying system hardware. Both the HAL and the PAL are groups of C++ and assembly functions called by the CLR. The HAL is what gets modified to support different processors and peripheral hardware. It contains some boot strap code, as well as, the drivers that make up the system.

Although the CLR HAL code is configured to boot directly, the EDK has an independent boot loader known as TinyBooter that boots the hardware and loads the CLR into memory. The Tinybooter is used to update the CLR code.

Hardware Layer - The hardware layer contains the microprocessor and other circuitry, and for the EDK this would be the iPac-9302.

Besides the iPac-9302, the .NET Micro Framework SDK includes support for an emulator that runs in Windows. You can develop, test, and debug applications before testing on actual hardware. The emulator is typically used when the target hardware is in design and development and not available, thus giving the software developers an opportunity to start managed code application development in advance of the hardware. Since you have a fully functional hardware platform, application development and debug can be done directly on the iPac-9302.

1.3 Real-time multitasking Kernel vs. CLR

If .NET MF was a real-time embedded kernel like a uCOS or a ThreadX, typically at this point we would introduce future topics for discussion such as priority levels, process spaces, memory management, etc. Threads and priority levels are supported, but not to the extended of other Real-time kernels. Knowing what the CLR is capable of and not capable of will help guide application development. We discuss these details a little later.

In the embedded market, there is a spectrum of operating systems and kernels available for embedded development. Many people may try to classify .NET MF at the end of the kernel spectrum, but .NET MF is NOT real-time and not really a kernel. A better description for .NET MF is an execution engine. It is important to think of .NET MF as an embedded CLR or managed code engine. Even though .NET MF doesn't feature real-time deterministic capability, its performance is fast enough for many small embedded systems. Chapter 4 will discuss the inner-processing mechanics.

1.4 About this Guide and Target Audience

The EDK is intended to reach a wide audience from students to hobbyists to professional developers. The examples provide a basic foundation for writing applications for .NET MF.

For those who have used a previous version of the guide, you will notice some changes in this revision. First, this revision supports .NET Micro Framework V4.0. There are some API changes to the serial port and Ethernet have been added. Second, we have removed the emulation chapter since having the real hardware is better than emulation. Third, we have moved information about the iPac-9302 to its own chapter. Forth, Extended Weak References has been moved to the appendix. Fifth, the chapters have been reordered slightly to improve information flow from application mechanics to hardware control. Finally, we have updated the various examples for .NET MF V4.0.

As before, this document covers the basic information regarding the CLR, C#, and .NET. There are other books available that dive deeper into these subjects. Even though these books cover coding on the desktop, the basic concepts are all the same. In some cases you can take the examples and run them in .NET MF.

This chapter serves as an introduction to the .NET MF and EDK. More complete .NET MF can be found in the online help documentation of the .NET MF SDK. The following chapters dive into different features of what can be done in the EDK.

- Chapter 2 covers the iPac-9302 features and setup. It also introduces some of the utilities for use with .NET MF and for upgrading firmware on the platform.

- Chapter 3 covers an overview of the .NET Micro Framework SDK and discusses how to write and download applications to the iPac-9302.

- Chapter 4 discusses the application/CLR processing loop and thread priorities available in .NET MF. A flash data storage technique is also discussed.

- Chapter 5 looks at the GPIO class and the GPIOs available on the iPac-9302. There are several exercises that address input, output, polling, and interrupts.

- Chapter 6 introduces the SPI class and how to interface to the SPI port. The exercises show how to connect to a 2x16 SPI LCD display and 4x4 keypad.

- Chapter 7 covers the second serial port that is part of the EDKPlus.

- Chapter 8 discusses the different PWMs supported on the iPac-9302 and the managed code driver that provides control of the PWMs.

- Chapter 9 covers the ADC and the managed code driver that provides read accesses to the data from the 5 ADC channels.

- Chapter 10 discusses Ethernet and creating applications with the socket class.

- Chapter 11 covers other miscellaneous topics and ideas to help with your coding.

If you are already familiar with .NET MF, then you can skip to different chapters, but the guide has been designed to be read from beginning to end. Each chapter builds upon the preceding chapters.

1.5 System Requirements for V4.0

.NET Micro Framework V4.0 now provides support for the Visual C# Express 2008.

1.5.1 Basic system Requirements

The basic requirements for the development machine are as follows:

- Intel Pentium 4 or higher
- 512MB of RAM, the more RAM the better.
- Windows XP Pro SP2 or higher
- Visual Studio 2008 with SP1 or higher, or Visual C# Express 2008 (Free Download)
- RS-232 Serial support used for downloading and debugging applications.
- .NET Micro Framework V4.0 SDK – free download from Microsoft.

Some of the exercises will require hardware that is not included in the kit. Appendix C has a list of these items.

1.5.2 USB Dongle Recommendations

Application development is performed over serial ports. Support for the RS-232 port is a rarity in modern PCs; especially laptops. There are USB-to-Serial dongles available, but not all are created equal. We recommend that you find a dongle that supports the Prolific USB-to-RS-232 IC chip (PL-2303 or better, http://www.prolific.com.tw). Here are some vendors that support this IC chip:

- http://usbgear.com/USBG-RS232-P72.html

- http://usbgear.com/765162.html

- http://www.sealevel.com/product_detail.asp?product_id=1631&Ruggedized%5FUSB%5Fto%5FRS%2D232%5FDB9%5FSerial%5FInterface%5FAdapter%5F

- http://www.beaglesoft.com/232usb.htm

1.6 Website

Every effort has been made to assure the accuracy of this development guide, but mistakes can fall through the cracks and Internet addresses can change. Please check the SJJ website for updates (www.sjjmicro.com), and please post a question to our contact / feedback form if you have any questions, suggestions, or require consulting services for your project.

2 iPac-9302 Setup

The iPac-9302 was specially designed for the .NET MF by EMAC, INC – www.emacinc.com. There are plenty of I/O options to build a variety of applications. The board itself meets the PC/104 dimensions so you can take advantage of PC/104 power supplies and enclosures.

A majority of the port signals are on 3 header blocks, while serial connections have their own separate 10 pin headers. There are two LEDs on board. The Red LED is a boot indicator not available for program control. The Green is available for program control through the GPIO interface. There is more than enough RAM and FLASH (8MB/8MB) for application and data storage.

EMAC has created an updated revision to the iPac-9302 that adds a new jumper - JB5 – for use with HDR1. This change was made to RV2 boards. Check the silk screen on the board to make sure that you know the correct revision of board that you have. We will show the difference between the boards in this chapter. Please see the iPac-9302 User's Guide for more information.

This chapter will cover the setup of the iPac-9302, some of the utilities that will be used, and supported changes for .NET MF V4.0. **Please read this chapter carefully as there are changes to both hardware and software development.**

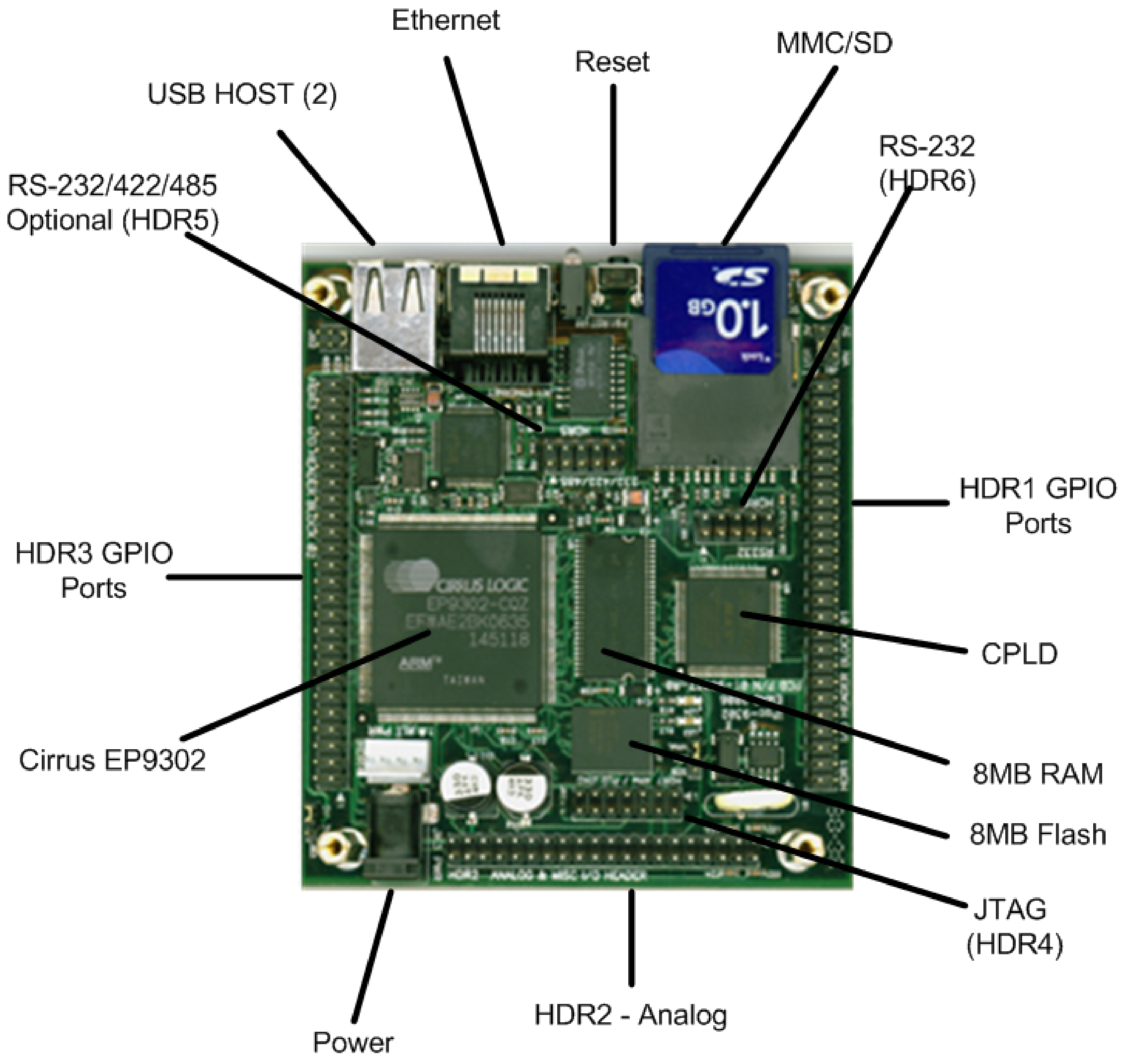


Fig 2.1 - iPac-9302 RV1

2.1.1 iPac 9302 Features

The following are the features of the iPac-9302:

- PC/104 Dimensions of 96 mm x 90 mm (3.77" x 3.54")
- 1 RS232 (HDR6), a second RS232/422/485 serial port (HDR5) is available on boards that come in the EDKplus.
- 1 10/100 Base-T Ethernet port
- 2 USB 2.0 host ports (not support by .NET MF)
- 5 channels of 12 bit A/D (0 to 3.3V)
- Internal Real time clock/calendar (no battery backup)
- A 50 pin header with 16 processor GPIO lines and 8 PLD output GPIO lines.
- A 50 pin header with 8 PLD 5 volt tolerant GPIO lines, 8 PLD output GPIO lines, and 8 PLD High Drive output GPIO lines.
- 2 PWM I/O lines from the CPLD and 1 PWM from the processor.
- SPI / AC97 / I2S options - .NET MF uses SPI only.

- External Reset Button provision and red & green Status LEDs
- 8 MB of External Flash
- 8 MB External SDRAM
- Battery backed RTC
- MMC/SD hot-swap socket*

2.1.2 Jumper Settings

There are 4 major jumper settings on the iPac-9302 RV0 and RV1, and 5 jumpers on the RV2 boards.

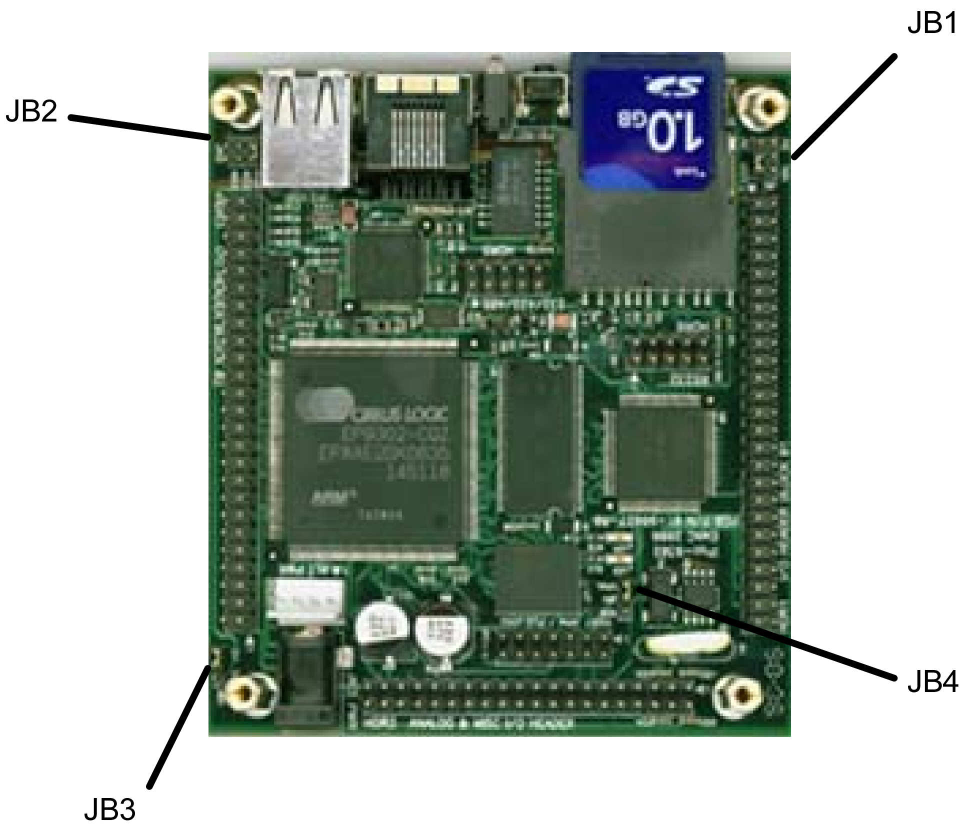


Fig 2.2 - Jumper Locations for RV1

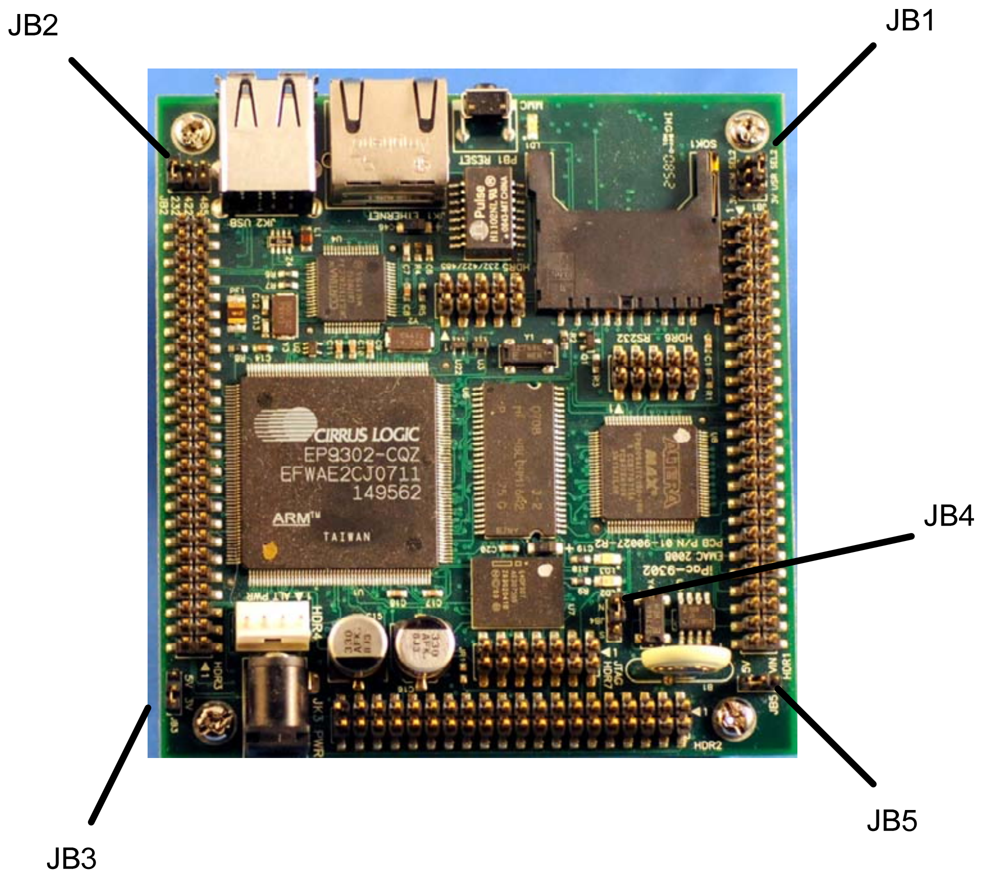


Fig 2.3 - Jumpers Locations for RV2

2.1.2.1 JB1 – HDR1 Pin 49 Voltage Selection and High Drive Pull-up Selection jumper for RV1 boards

The following diagram and table show the information:

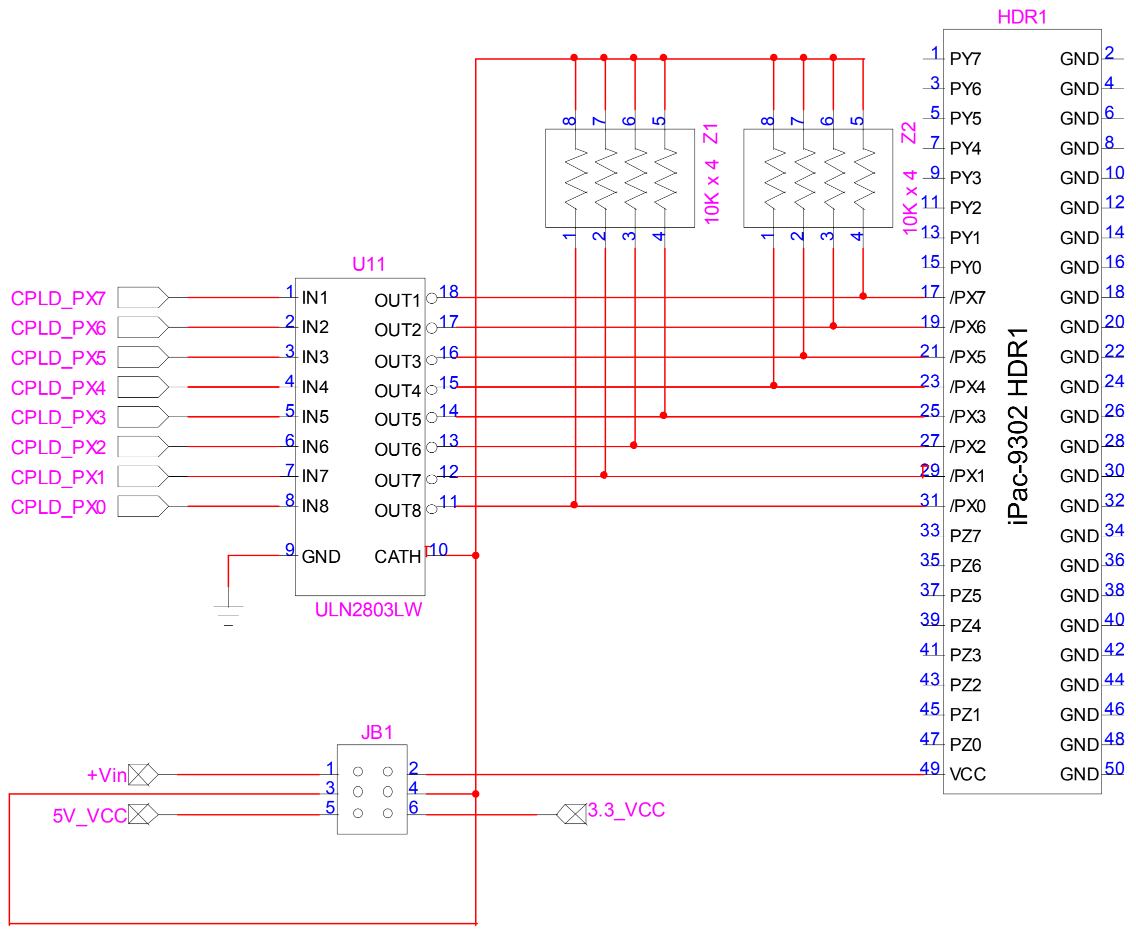


Fig 2.4 - Jumper Block JB1 for RV1 Boards

Jumper		Configuration
1	2	Vin (HDR4, pin4) applied to HDR1 pin 49, no pull-up voltage (open).
1	3	Vin (HDR4, pin4) applied to pull-ups, no connection to pin 49.
2	4	User provided voltage on HDR1 pin 49 applied to pull-ups.
3	5	+5 volts applied to pull-ups, no connection to HDR1 pin 49
4	6	+3.3 volts applied to pull-ups, no connection to HDR1 pin 49.

Table 2.1 - JB1 Jumpers for RV1

WARNING *- Do not connect pins 5 & 6 or you will short out the supply and void the warranty of the board.*

2.1.2.2 RV2 JB5 and JB1 Jumper Options

There are a few changes made in iPac-9302 RV2. First, the pull up resistors are at a higher resistance value (51k ohms). Second, the addition of JB5 helps to avert connecting the wrong voltage and shorting the board.

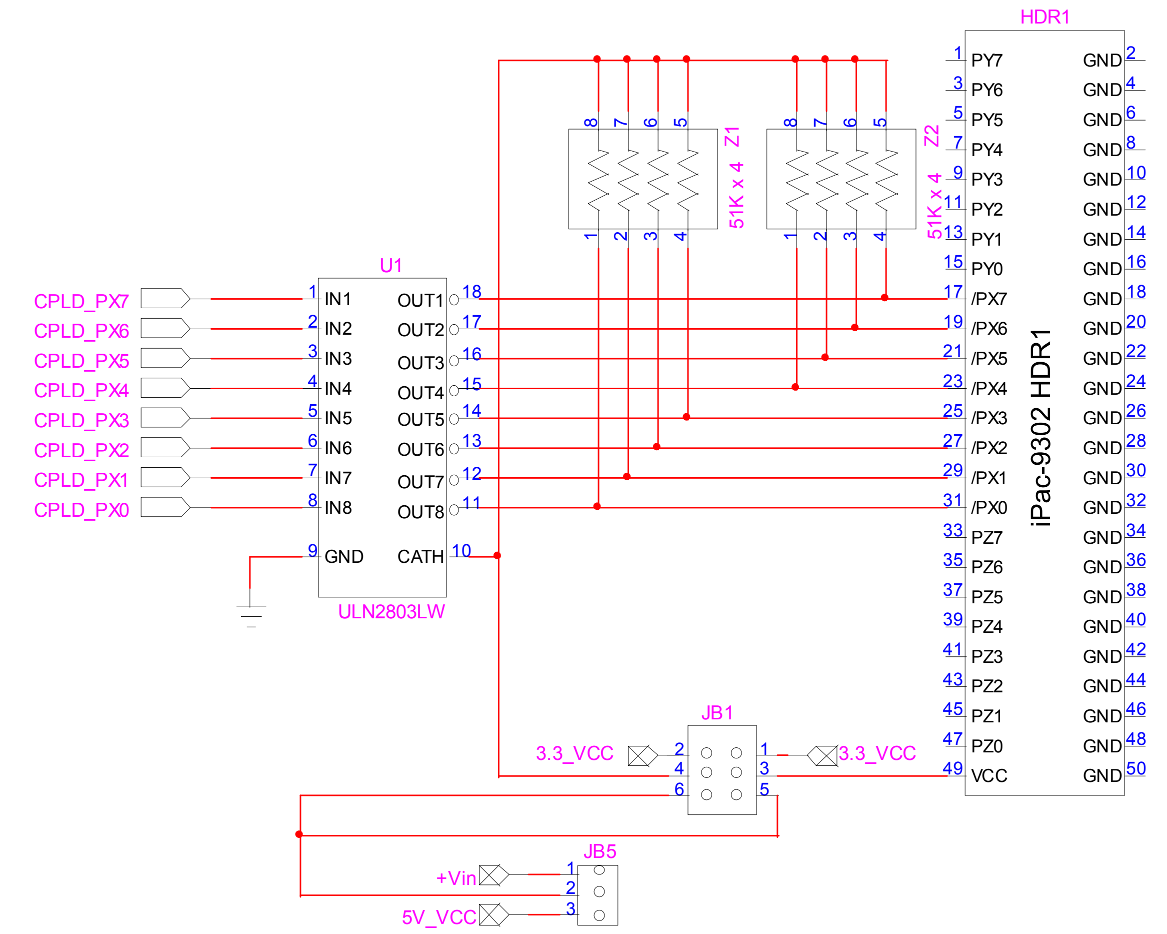


Fig 2.5 - Jumper Block JB1 and JB5 for RV2 Boards

Table 2.2 – shows that JB1 has split the jumper/voltage selection for HDR1 Pin 49 and the pull up resistors.

Jumper		Configuration
1	3	3.3V on Pin 49 VCC on HDR1
3	5	Voltage on HDR1 Pin 49 VCC is determined from JB5
2	4	3.3V applied for pull up resistors
4	6	Pull up resistor voltage determined from JB5

Table 2.2 - JB1 Jumper settings for RV2

Jumper		Configuration
1	2	Vin is selected
2	3	5V is selected

Table 2.3 - JB5 Jumper Selections

Basic examples of the JB1 and JB5 selections:

- If JB1 has a jumper on pins 1 and 3, 3.3V will be available on HDR1 Pin 49.

- If JB1 has jumper on pines 3 and 5, and JB5 has a jumper on 2 and 3, 5V will be available on HDR1 Pin 49.

2.1.2.3 JB2 – UART1 (HDR5) Serial Selection Jumper (Optional on some boards)

Jumper		Setting
1	2	RS-232 Serial
3	4	RS422 Full Duplex (transmitter always on)
5	6	RS485

Table 2.4 - JB2 Jumpers

2.1.2.4 JB3 – I/O header (HDR3, pin49) Voltage Selection Jumper

JB3 sets the voltage for the pin to 3.3V or 5V for use with external devices.

Jumper		Voltage
1	2	+3.3V
2	3	+5V

Table 2.5 - JB3 Jumpers

2.1.2.5 JB4 Boot option Jumper

Jumper		Configuration
1	2	Serial Boot through UART 1 (See section 2.8 on the Cirrus boot utility)
2	3	Normal Boot from Flash.

Table 2.6 - JB4 Boot option Jumpers

2.2 Exercise 2.1: Hardware and EDK Setup

Since you are probably eager to power up your new hardware and download an application, let's setup the hardware for use with the various exercises that we will be performing in this guide.

1. Get the standoffs out of their package.
2. Remove the iPac-9302 from its static bag. Be careful not to get the board near any static source that could damage the board (using a properly grounded, conductive wrist-strap is a good antistatic practice when handling circuit boards and electronic components).
3. Screw on 1 standoff to each of the 4 corner holes.

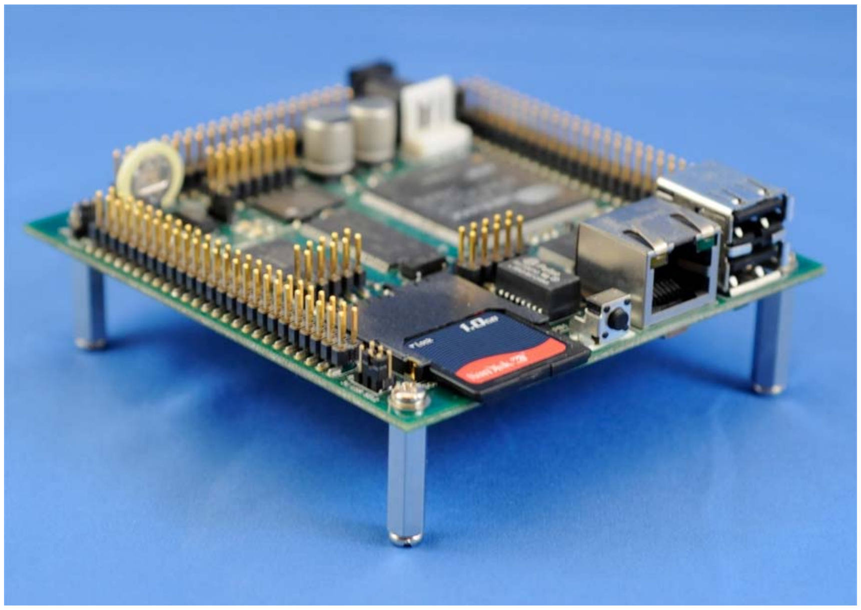


Fig 2.6 - iPac-9302 with Standoffs

4. Set the jumpers to the following configuration:

Jumper	Setting
JB1	For RV1: Pins 1 & 2 (Vin HDR1, pin49) For RV2: Pins 3 and 5 (voltage for HDR1 pin 49 is selectable by JB5)
JB2 (optional)	Pins 1 & 2 RS-232 (not supported in the EDK version)
JB3	Pins 2 & 3 (+5V HDR3, pin 49)
JB4	Pins 2 & 3 (Normal boot)
JB5 (RV2 only)	Pins 2 and 3 (5V)

Table 2.7 - iPac-9302 .NET MF Jumper Configuration

5. Connect the serial ribbon cable to header 6 (HDR6). Be sure to align the color stripe to pin 1 on HDR6 and make sure that all pins on the header align with the cable connector.

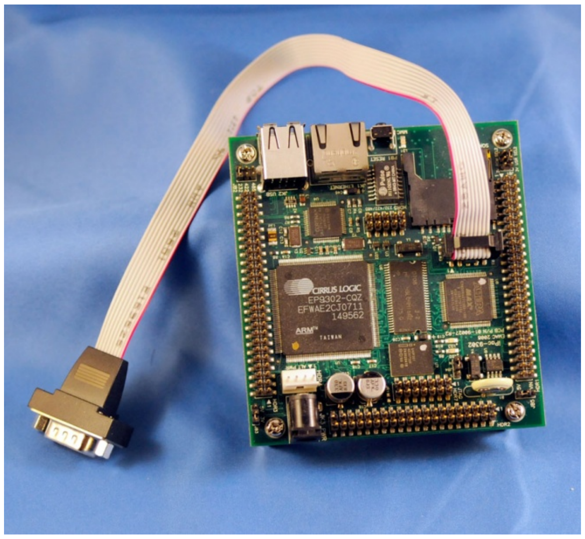


Fig 2.7 - iPac-9302 with COM1 Serial Cable Attached

6. Connect one end of the Null modem cable to the DB-9 of the ribbon cable and the other end to the PC serial port.
7. Connect an Ethernet cable to the Ethernet port if you plan to use DHCP. Otherwise, use MFDeploy to set a static TCP/IP address when the Ethernet cable is not connected.
8. (Optional): if you have a fully populated iPac-9302, you may want to connect another ribbon cable to the COM2 port 10-pin header 5 (HDR5), again making sure that the color stripe on the cable aligns with pin 1 and that all pins of the header align with the cable connector.
9. Make sure that you have installed Visual Studio or Visual C# express and that you have installed .NET Micro Framework V4.0 SDK. Insert the EDK CD in to your development PC.
10. You will be asked to run the batch file that launches the installation webpage, click **OK** or select the **autorun** to continue.
11. The setup page launches the browser. The menu on the left lists the different documentations and tools for use with the EDK. All the documents require Adobe® Reader®. If you are reading this document, then you already have the reader installed. You can save one copy of the Step-By-Step Guide to your hard drive.

Menu Item	Description
Step-By-Step Dev. Guide	This guide which provides detailed information on the .NET Micro Framework running on the iPac-9302
iPac-9302 Documentation	Important information regarding the iPac-9302 hardware. You should read this documentation before connecting anything to the iPac-9302
Chapter Exercise	ZIP file that contains the solutions and tools needed for the different exercises in the development guide.
Utilities	Contains an installer for the SJJ_COMM Lite serial communications application. Click on this menu and run the setup to install SJJ_COMM Lite
EULA and Warranty	Legal documents pertaining to the use of the EDK
www.sjjmicro.com	Link to SJJ Embedded Micro Solutions Website

Table 2.8 - EDK CD Contents

12. Click on the **Chapter Exercises** and save the ZIP file to your local hard drive.

13. Extract the ZIP file to decompress the chapter solutions, SJJ Hardware provider DLL, and Visual Studio template.
14. Optional - Click on the Utilities menu item to install SJJ_COMM Lite.

***Note**: Some of the exercises in this guide will require extra hardware that is not part of the kit. Appendix C contains the list of these hardware items.*

2.3 .NET Micro Framework v4 Port to the iPac-9302

The iPac-9302 has many built-in I/O devices. Not all of these devices are support in the .NET MF port. Some are not support because .NET MF itself doesn't have the capability, and others are future supported items.

2.3.1 What does the .NET MF port support on the iPac-9302

The first versions of the iPac-9302 / .NET MF port (V1.0 & V2.0) supported, GPIOs, interrupt GPIOs, debug serial out, SPI, and Flash storage.

Version 2.5 adds support for a second serial port for application use, analog-to-digital converters (ADC), Pulse Width Modulators (PWM), and Ethernet (TCP/IP and Sockets).

Version 4.0 R1 supports the same I/O as version 2.5. A planed Version 4.0 R2 could add support for SD card, battery backed RTC, and possibly application deployment and debug over Ethernet.

Feature	**.NET MF Release for the EDK**		
	V1.0 and V2.0	**V2.5**	**V4.0 R1**
GPIO	X	X	X
Interrupt GPIO	X	X	X
COM1 Debug Serial	X	X	X
COM2 EDKPlus	X	X	X
SPI	X	X	X
Flash Storage - EWR	X	X	X
ADC		X	X
PWM		X	X
Ethernet		X	X
Application development over Ethernet / COM1 freed up for applications			Future
SD Card Support			Future
Battery Back RTC – EDKPlus only			Future
USB Host	N/A	N/A	N/A

Table 2.9 - .NET MF Version and Support History of the iPac-9302

A time delay added in the registry for deploying applications was introduced in V2.5. The reason the delay was needed was for the Ethernet implementation. These keys can (and should) be removed with V4.0; more discussion on this later in the chapter.

NET MF doesn't currently have internal API support for USB Host interfaces, thus the USB host ports on the iPac-9302 are not available.

2.3.1.1 Boot Loader

There is a boot loader called TinyBooter that is on the flash chip. The boot loader will help with updating the CLR in the future.

After TinyBooter boots up, there is a boot delay before the CLR is launched. The delay allows interacting with TinyBooter before the CLR launches so that the CLR can be updated in the future. The boot delay is 20 seconds if MFDeploy is connected via the debug COM port, or 5 seconds if a terminal application like SJJ_COMM Lite is connected via the debug COM port. When using MFDeploy, the boot delay can be forcibly terminated early. How to terminate the boot delay in MFDeploy will be described later. During a normal boot operation, the iPac-9302's boot hardware will detect the boot loader, TinyBooter, and execute it. TinyBooter will do basic hardware configuration and then listen for commands on the debug serial port during the boot delay period. If it receives commands, it will respond. One such command is to flash a new version of the CLR, and another is to erase the managed code and data areas. This will be discussed in more detail, later. Typically, no commands are sent to TinyBooter, so it passes control to the CLR. CLR looks for a managed code application to run. If no managed code application is found, CLR, listens to the debug serial port for commands, such as a connect command from Visual Studio. If a managed code application is found, it loads it and passes control to it.

2.3.1.2 Flash Memory Map

TinyBooter, CLR, and managed code applications and data are all stored in flash memory. Referring to Table 2.10, the TinyBooter image is stored in the "Bootstrap" area of flash memory. When TinyBooter starts up, a decompression utility runs out of flash that decompresses the compressed TinyBooter image to RAM and jumps to the decompressed image. At this point, TinyBooter runs from RAM.

Sectors	Start	Size	Usage
0	0x60000000	0x00020000	Bootstrap
1	0x60020000	0x00020000	Configuration
2-23	0x60040000	0x00020000 (each)	Code
24-31	0x60300000	0x00020000 (each)	Deployment
32-47	0x60400000	0x00020000 (each)	Storage
48	0x60600000	0x00020000	Logging
49-62	0x60620000	0x00020000 (each)	Reserved
63-66	0x607E0000	0x00008000 (each)	Reserved

Table 2.10 - Flash Memory Map

CLR and its configuration data are stored in the "Configuration" and "Code" sections of flash memory. When CLR is running, it has to be able to write to flash memory. When CLR is writing to flash memory, it cannot be executing its instructions from flash memory at the same time; therefore, when control is first passed from TinyBooter to CLR, CLR relocates a certain number of its routines from flash memory to RAM, where they will be accessed during normal execution. Routines such as those that write to flash memory are relocated to RAM and executed from RAM during normal CLR operation. With the exception of routines like those, CLR runs from FLASH.

Managed code modules and their associated data are stored in the "Deployment" section and "Storage" section of flash memory respectively. Managed code modules and data are almost exclusively flash memory entities.

There is also a "Logging" section that has proprietary use by Microsoft. Logging is not supported in the EDK.

Finally, not all of the flash memory has been configured for use in this version of the EDK. There are "Reserved" sections of memory that can be optionally included for use at a future time.

There is 0x100000 bytes (1 MByte) each of managed code application space, and data storage space. As you can see there is a lot of room for storing program and data.

2.4 MFDeploy Utility and Setting Static TCP/IP

MFDeploy is a combo graphical GUI and command line interface program that is used to update the CLR and erase application deployment sectors in flash. If the program is started without parameters, a graphical GUI interface starts up. It takes several seconds to launch, so be patient when launching MFDeploy.

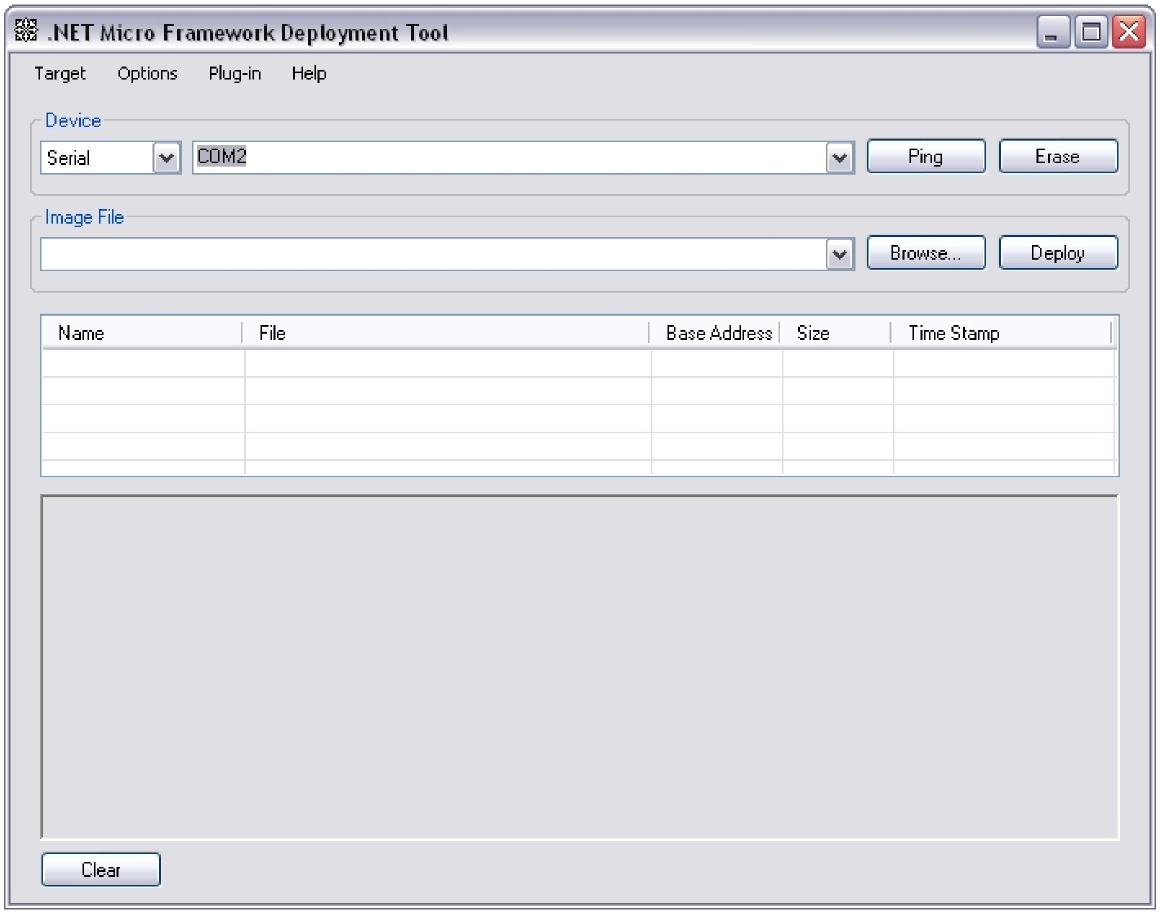


Fig 2.8 - MFDeploy

2.4.1 Command line options

You can perform some of the functions from the command line.

Usage: MFDeploy [COMMAND[:CMD_ARGS]] [/I:INTERFACE[:INTEFACE_ARGS]] [FLAG]

Where COMMAND is:

Command	Description
Erase	Erases deployment sectors.
Deploy:<image file>[;<image file>]*	Deploys the given SREC (.hex) files to the device.
Ping	Verifies connection with device.
Reboot[:Warm]	Reboots the device. Warm option only restarts the CLR.
List	Lists available ports (USB and Serial).
Help	Displays this screen.

Table 2.11 - MFDeploy Commands

Where INTERFACE is:

Interface Type	Description
Serial:<port_num>	Identifies serial port number for the device.
USB:<usb_name>	Identifies USB port name for device.
TCPIP:<ip_addr>	Identifies TCP/IP address for device.

Table 2.12 - MFDeploy Interface Types

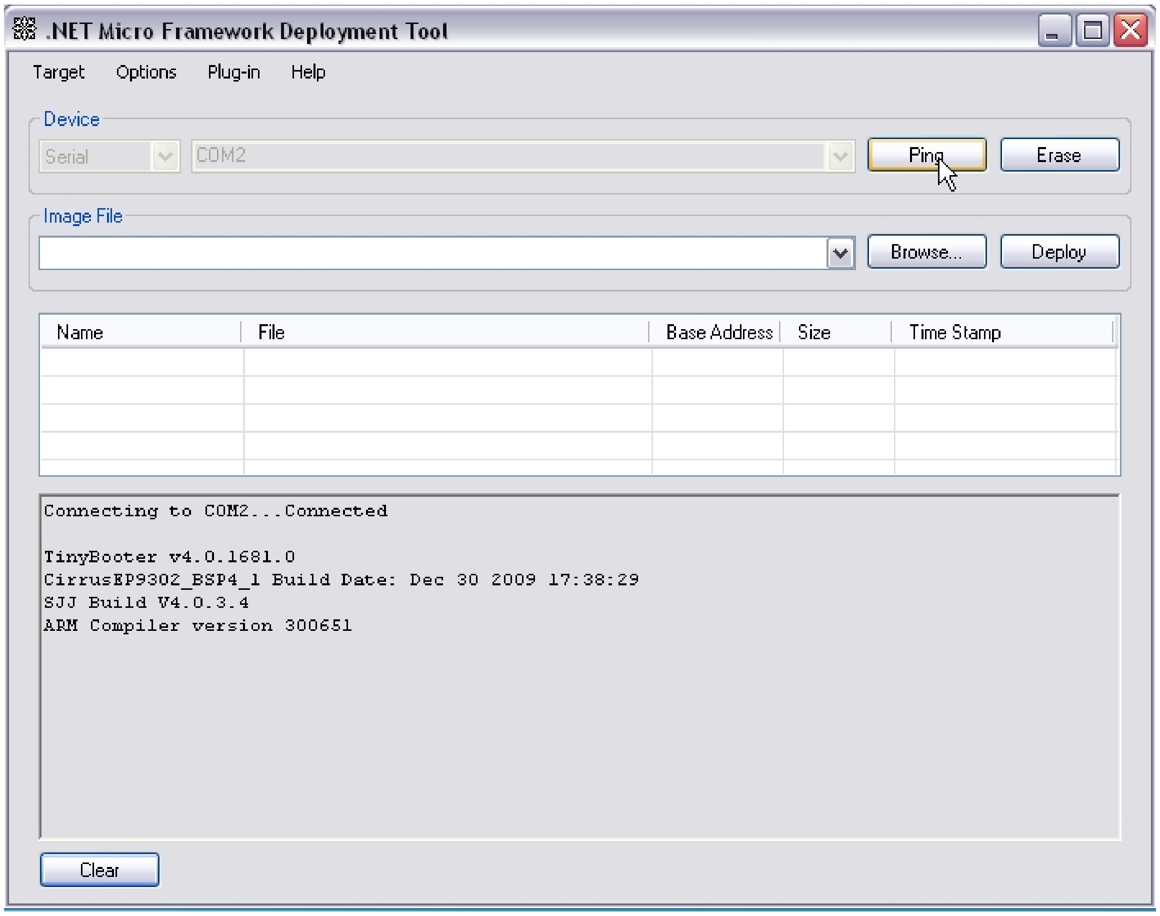
***Note**: If two interfaces are given, the second is assumed to be for Tinybooter.*

The graphical GUI version also comes with some extensible debug plug-ins that provides further capabilities for development. We will primarily be using the graphical GUI version to interact with the iPac-9302 in these exercises.

2.4.2 Clearing the previous application

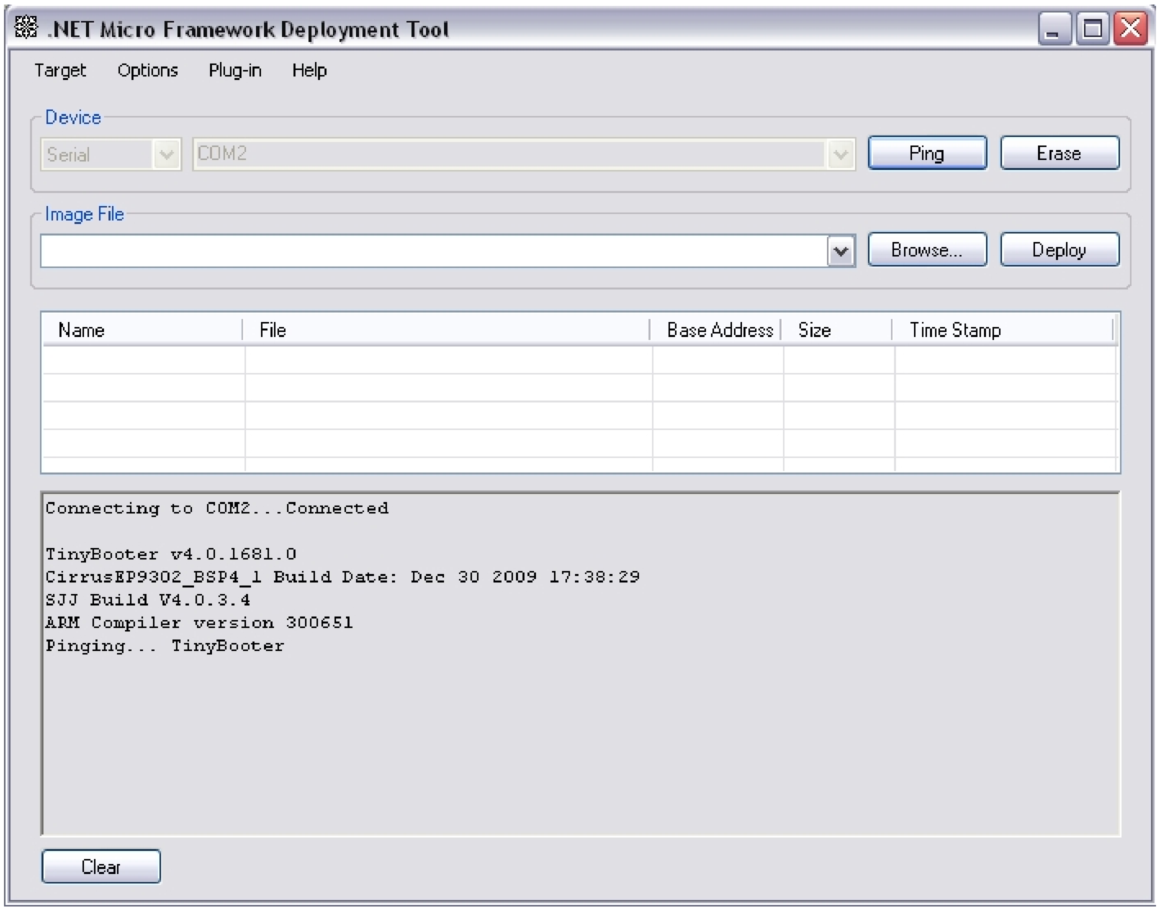
If you are having trouble downloading an application, you may have to erase the deployment area before attempting to deploy a new application build. Keep in mind that the CLR is not real-time, nor preemptive, so if you have an application running that is dominating the CLR's resources, it may not be able to respond to connect and deploy commands from Visual Studio. This is especially true if you have an application with runtime errors that have not be debugged, yet. The solution is to erase the current application using TinyBooter, and then rerunning the CLR without the competing application. To erase a managed code application, do the following:

1. Power down the iPac-9302.
2. Make sure the serial cable is connected to the development PC and the debug port on the iPac-9302.
3. Open MFDeploy in graphical mode.
4. Select the appropriate COM port for you development PC. And from the dropdown select **Target->Connect**.
5. Power up the iPac-9302.
6. Watch the output window at the bottom of the MFDeploy form. When TinyBooter signs on and displays its version, click the **Ping** button to confirm that you are talking to Tiny booter.



```
Connecting to COM2...Connected

TinyBooter v4.0.1681.0
CirrusEP9302_BSP4_1 Build Date: Dec 30 2009 17:38:29
SJJ Build V4.0.3.4
ARM Compiler version 300651
```

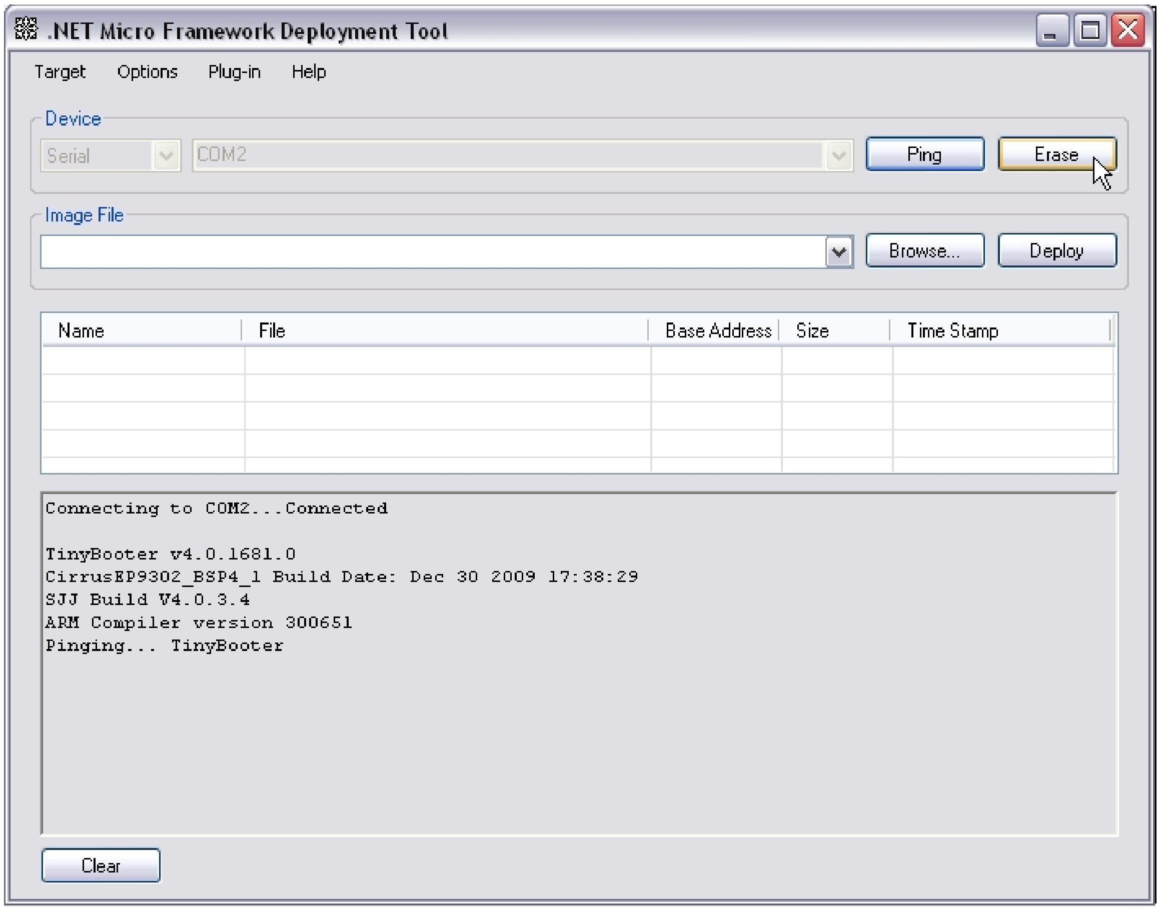
Fig 2.9 - MFDeploy Ping TinyBooter



```
Connecting to COM2...Connected

TinyBooter v4.0.1681.0
CirrusEP9302_BSP4_1 Build Date: Dec 30 2009 17:38:29
SJJ Build V4.0.3.4
ARM Compiler version 300651
Pinging... TinyBooter
```

Fig 2.10 – MFDeploy TinyBooter Ping Response

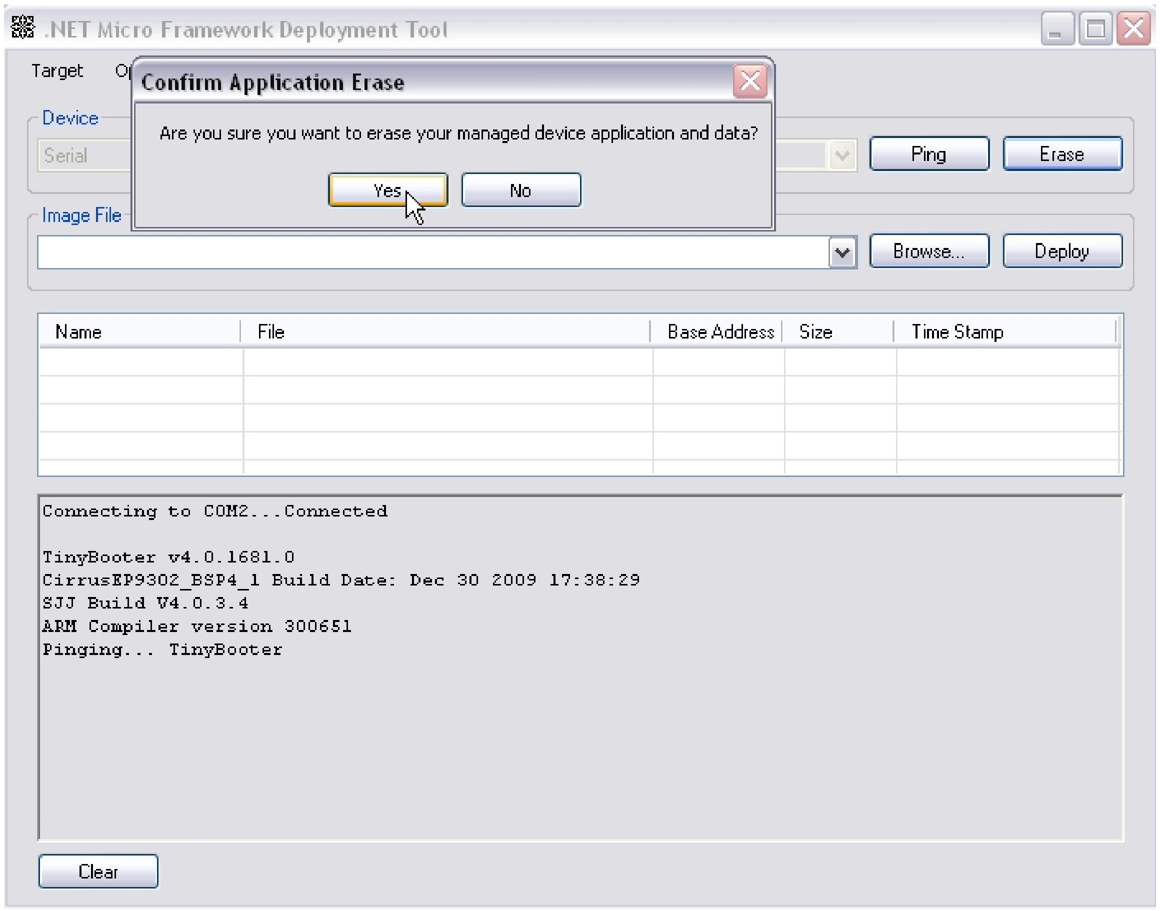
7. When you get a response from TinyBooter, click the **Erase** button.



```
Connecting to COM2...Connected

TinyBooter v4.0.1681.0
CirrusEP9302_BSP4_1 Build Date: Dec 30 2009 17:38:29
SJJ Build V4.0.3.4
ARM Compiler version 300651
Pinging... TinyBooter
```

Fig 2.11 - MFDeploy Select Erase Managed Code Application

8. MFDeploy will then give you a confirmation dialog. Click **Yes** in the confirmation dialog box. This will erase the application area in flash.

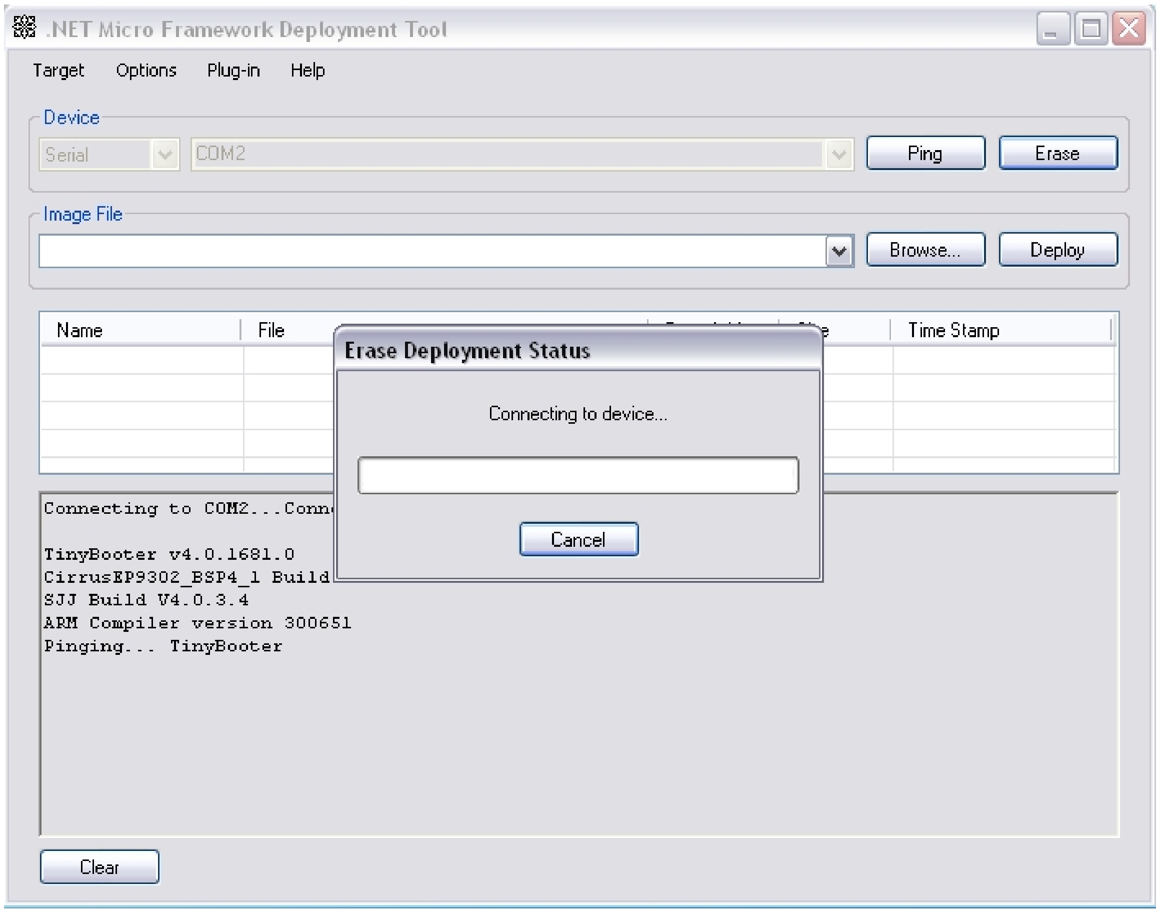


Are you sure you want to erase your managed device application and data?

```
Connecting to COM2...Connected

TinyBooter v4.0.1681.0
CirrusEP9302_BSP4_1 Build Date: Dec 30 2009 17:38:29
SJJ Build V4.0.3.4
ARM Compiler version 300651
Pinging... TinyBooter
```

Fig 2.12 - MFDeploy Erase Managed Code Confirmation

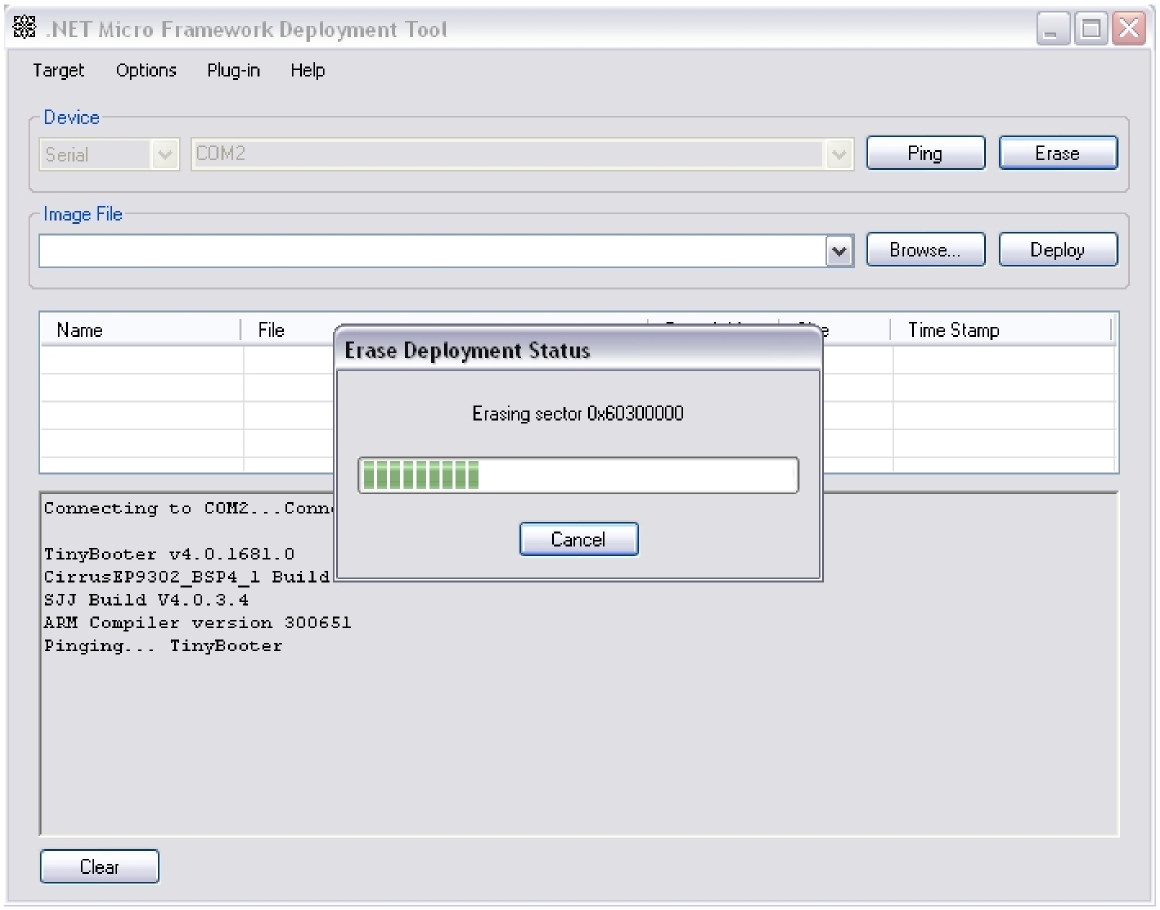
9. MFDeploy will show a status dialog showing connecting to the device.



```
Connecting to COM2...Conn
TinyBooter v4.0.1681.0
CirrusEP9302_BSP4_1 Build
SJJ Build V4.0.3.4
ARM Compiler version 300651
Pinging... TinyBooter
```

Fig 2.13 - MFDeploy Device Connection Status Dialog

10. After connecting to the device, MFDeploy will show an erase status dialog as it actually erases the managed code application and its data.



Erasing sector 0x60300000

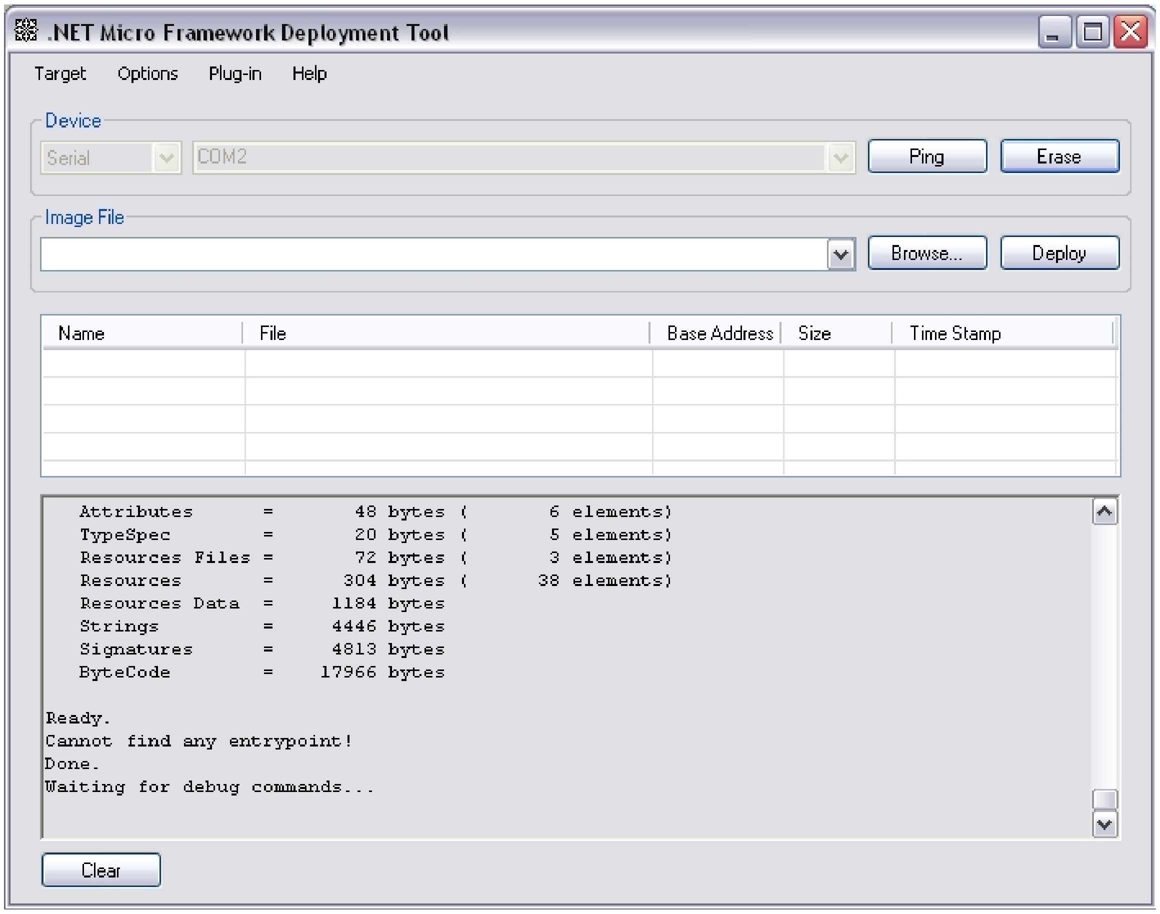
```
Connecting to COM2...Conn

TinyBooter v4.0.1681.0
CirrusEP9302_BSP4_1 Build
SJJ Build V4.0.3.4
ARM Compiler version 300651
Pinging... TinyBooter
```

Fig 2.14 - MFDeploy Application Erase Status Dialog

11. When the erase operation has completed and the status dialog box has closed, you can wait the 20 second MFDeploy boot delay, you can clear the bootloader flag (discussed next), or you can power cycle the iPac-9302 to boot the CLR.
12. Download the new application from Visual Studio.

NOTE: *If MFDeploy complains about access to the COM port, you might need to close Visual Studio. Visual Studio sometimes locks access to the COM port, and will not relinquish it.*

13. When the CLR loads, it will output version information, network configuration, and assembly information. With the managed code application area erased, there will be a delay, as the CLR searches for an application signature in the deployment area and then it will show that it cannot find an entrypoint and is waiting for debug commands.



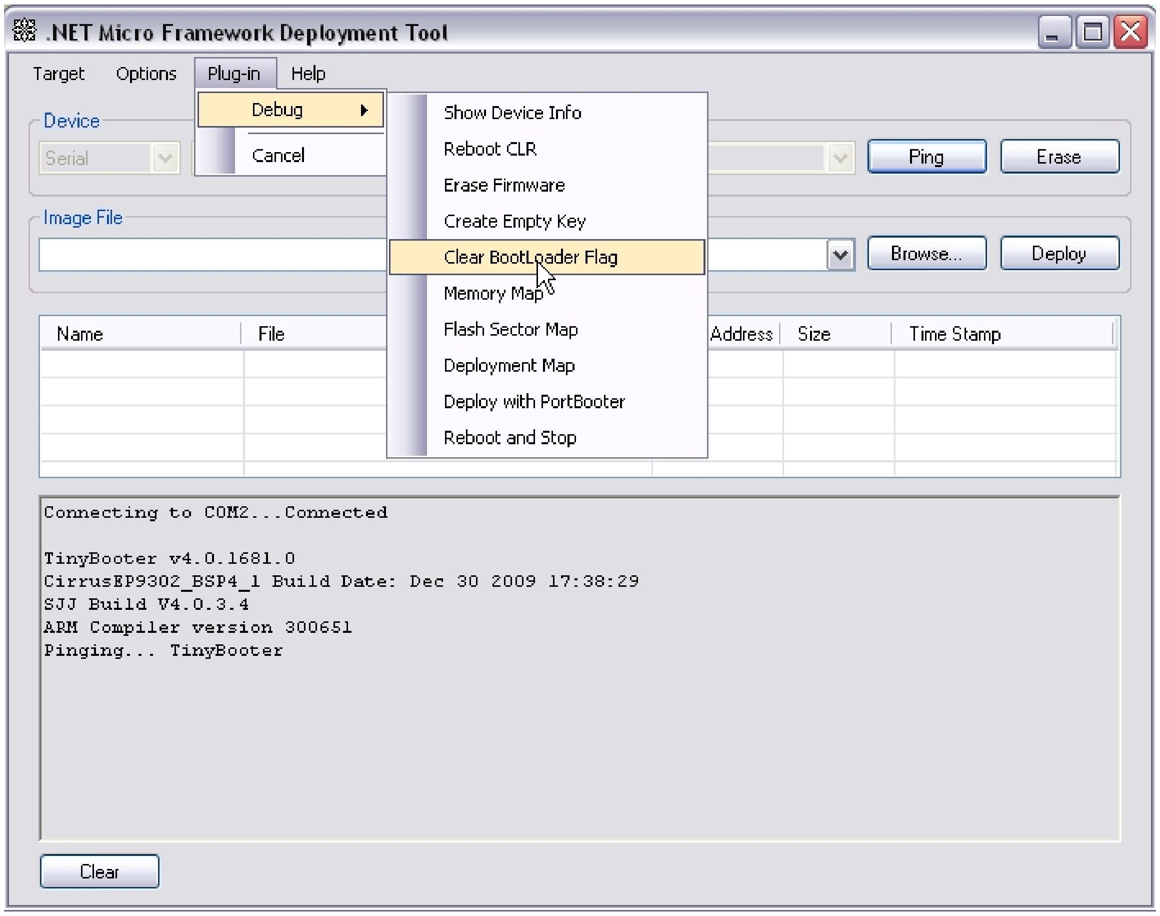
```
   Attributes      =       48 bytes (       6 elements)
   TypeSpec        =       20 bytes (       5 elements)
   Resources Files =       72 bytes (       3 elements)
   Resources       =      304 bytes (      38 elements)
   Resources Data  =     1184 bytes
   Strings         =     4446 bytes
   Signatures      =     4813 bytes
   ByteCode        =    17966 bytes

Ready.
Cannot find any entrypoint!
Done.
Waiting for debug commands...
```

Fig 2.15 - CLR Booted with No Managed Code Application

2.4.3 Clearing the Bootload Flag

When the iPac-9302 is booted with MFDeploy connected to the debug port, TinyBooter sets a bootload flag, that tells TinyBooter to delay loading the CLR. The time to wait is set to 20 seconds. After the 20 second delay, the bootloader flag is clear, and TinyBooter loads the CLR. MFDeploy has a Plug-in command to force the clearing of the bootloader flag, so that you can force TinyBooter to load the CLR without waiting the entire 20 seconds. To clear the bootloader flag:

1. Open MFDeploy in graphical mode.
2. Select the appropriate COM port for you development PC. And from the dropdown select **Target->Connect**.
3. Power up the iPac-9302.
4. Watch the output window at the bottom of the MFDeploy form. When TinyBooter signs on, from the dropdown select **Plug-in->Debug->Clear BootLoader Flag**.



```
Connecting to COM2...Connected

TinyBooter v4.0.1681.0
CirrusEP9302_BSP4_1 Build Date: Dec 30 2009 17:38:29
SJJ Build V4.0.3.4
ARM Compiler version 300651
Pinging... TinyBooter
```

Fig 2.16 - MFDeploy Select Clear BootLoader Flag

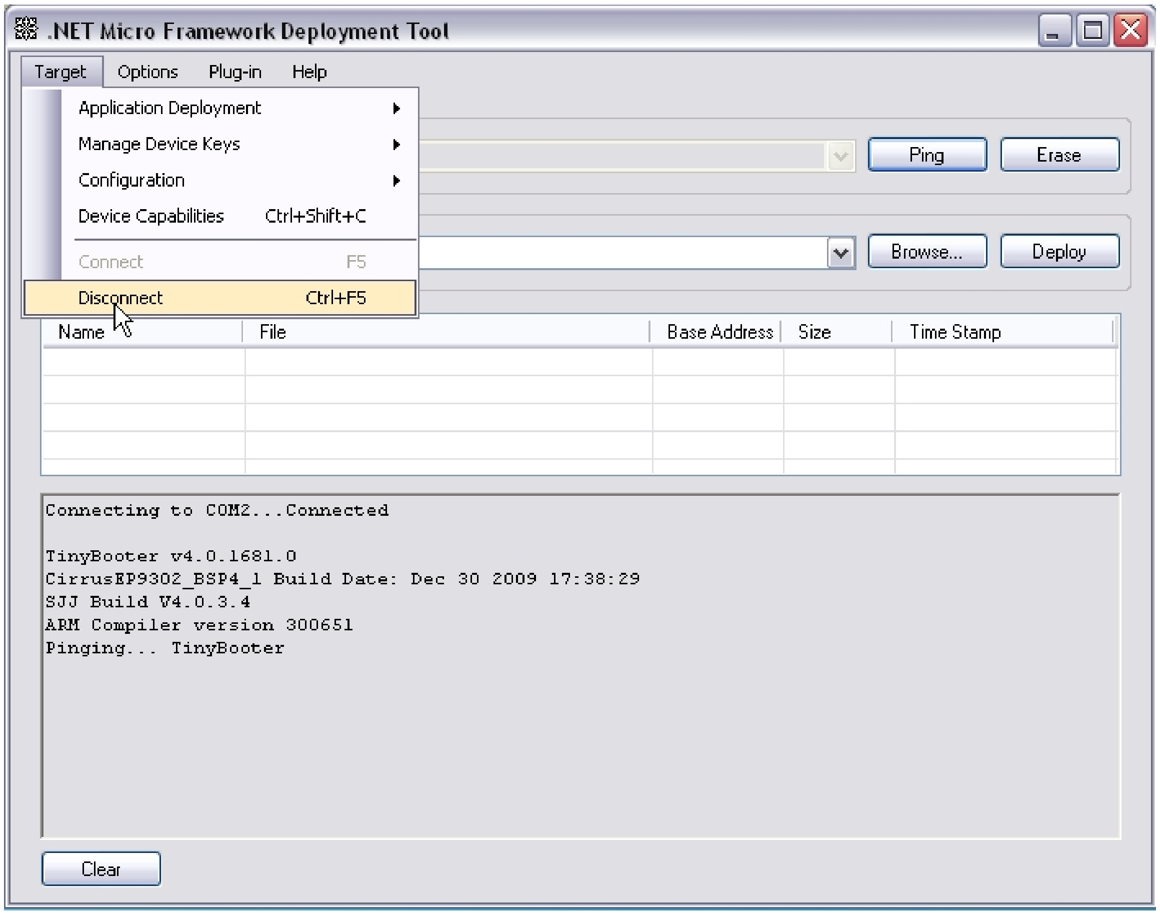
5. After selecting Clear BootLoader Flag, TinyBooter will terminate the boot delay and launch the CLR.

2.4.4 Setting Static TCP/IP

The default Ethernet configuration for the EDK as-shipped is to use DHCP to obtain an IP ADDRESS. As .NET MF is loading, the boot sequence looks to get an IP address using THE DHCP discovery sequence. The device will send out a request and it will wait 8 seconds for a response from a DHCP server. If there is no response, the device will repeat the request and and wait up to 5 times, before it will fail and complete the boot process. The delay will take a long time if the Ethernet cable is not connected or a DHCP server is not available. If you want to run without Ethernet or DHCP services, to get around the delay, you will want to set a static IP address. MFDeploy gives you the capability to select between DHCP or static IP address mode. MFDeploy can be used to set the network configuration via TinyBooter or the CLR. The MFDeploy commands are the same for either and the results are the same. The only difference is that MFDeploy must be disconnected to set the network configuration from TinyBooter, but from the CLR you can be connected. To change the network configuration to static IP via TinyBooter, do the following:

1. Power down the iPac-9302
2. Make sure the serial cable is connected to the development PC and the debug port on the iPac-9302.
3. Open MFDeploy in graphical mode.

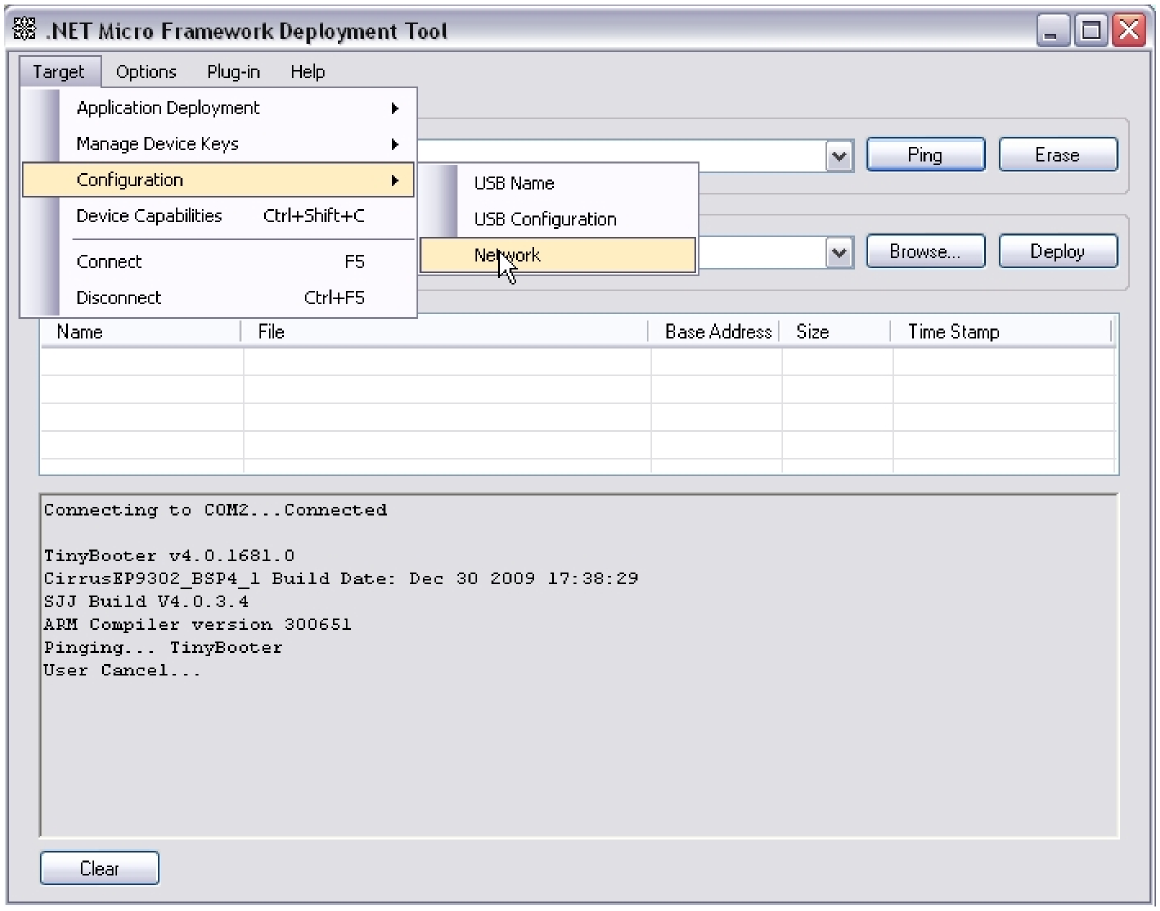
4. Select the Serial device and the COM port that the iPac-9302 debug serial port is connected to.
5. From the dropdown select **Target->Connect** so you can observe the boot-up process and confirm that TinyBooter has booted up.
6. Power up the iPac-9302.
7. When TinyBooter logs on, click on **Ping** to verify that you are talking to TinyBooter.
8. From the dropdown select **Target->Disconnect**.



```
Connecting to COM2...Connected

TinyBooter v4.0.1681.0
CirrusEP9302_BSP4_1 Build Date: Dec 30 2009 17:38:29
SJJ Build V4.0.3.4
ARM Compiler version 300651
Pinging... TinyBooter
```

Fig 2.17 - MFDeploy Target Disconnect

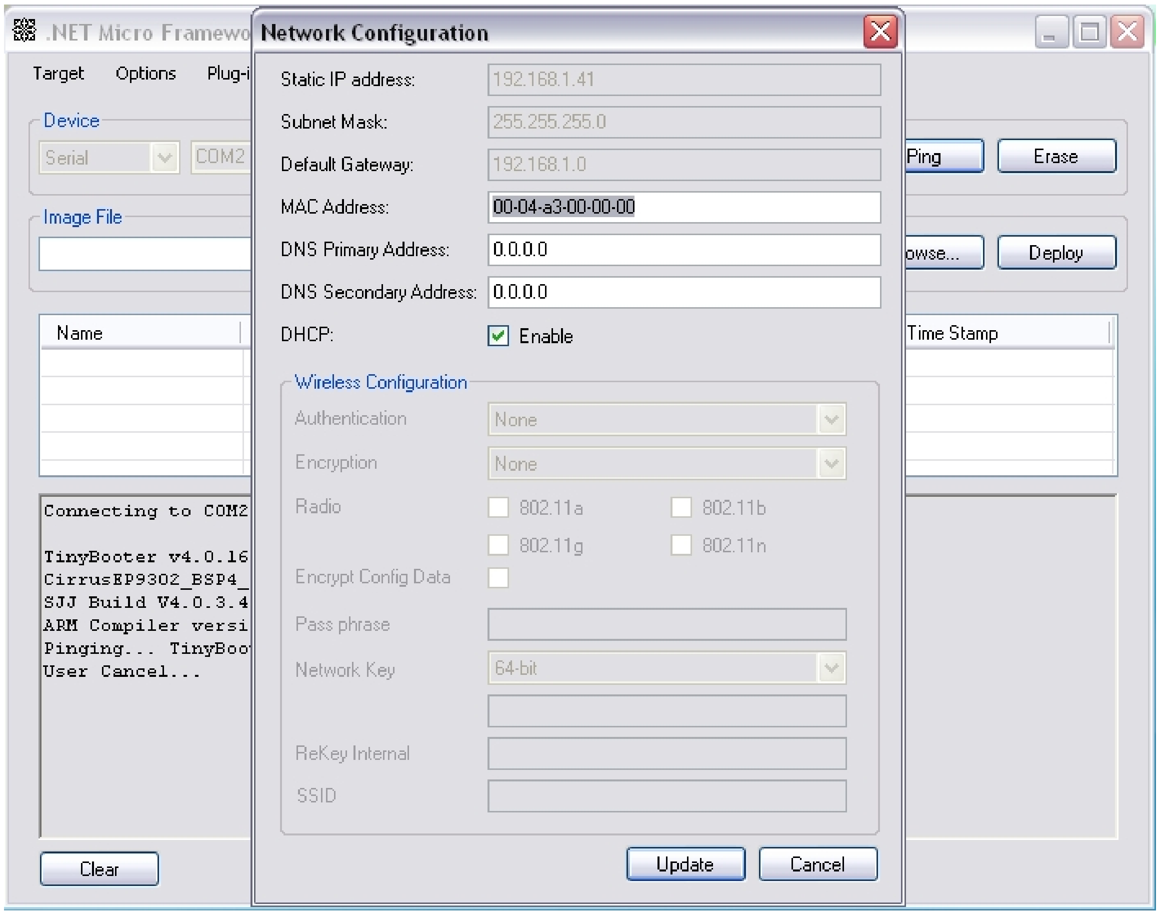
9. ,Select **Target->Configuration->Network**.



```
Connecting to COM2...Connected

TinyBooter v4.0.1681.0
CirrusEP9302_BSP4_1 Build Date: Dec 30 2009 17:38:29
SJJ Build V4.0.3.4
ARM Compiler version 300651
Pinging... TinyBooter
User Cancel...
```

Fig 2.18 - MFDeploy Configuration Nework Select

10. MFDeploy will open a dialog for the network configuration settings.



Serial COM2

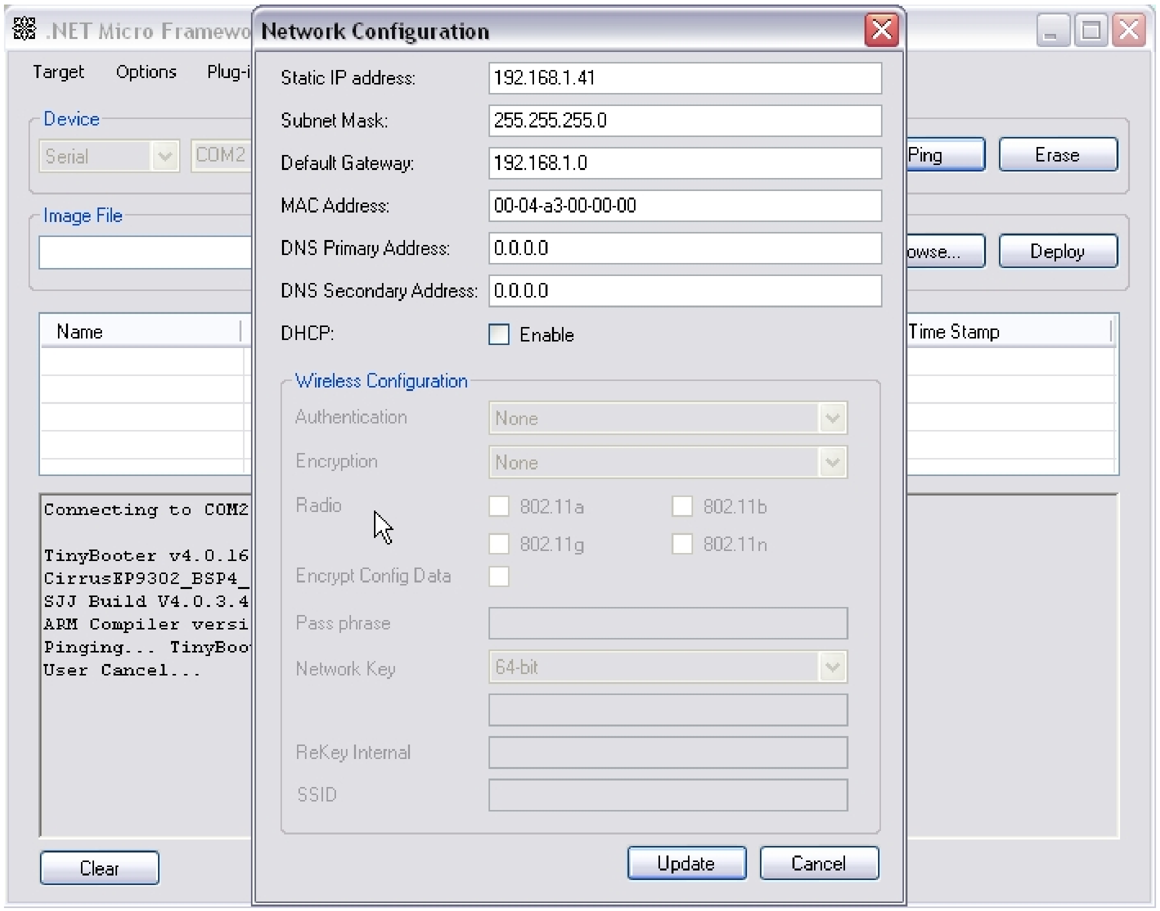
```
Connecting to COM2

TinyBooter v4.0.16
CirrusEP9302_BSP4_
SJJ Build V4.0.3.4
ARM Compiler versi
Pinging... TinyBoo
User Cancel...
```

Static IP address:	192.168.1.41
Subnet Mask:	255.255.255.0
Default Gateway:	192.168.1.0
MAC Address:	00-04-a3-00-00-00
DNS Primary Address:	0.0.0.0
DNS Secondary Address:	0.0.0.0
DHCP:	✓ Enable
Authentication	None
Encryption	None
Radio	☐ 802.11a ☐ 802.11b ☐ 802.11g ☐ 802.11n
Encrypt Config Data	☐
Pass phrase	
Network Key	64-bit
ReKey Internal	
SSID	

Fig 2.19 – Network Configuration Settings

11. To set a static IP address, uncheck the DHCP: Enable box, and you will then be able to set a Static IP address, subnet mask, default Gateway, etc.

Note: *the MAC address shown in the configuration settings dialog is not the devices actual MAC address and changing the value will not change the device's MAC address. The MAC address is set by the manufacturer and is not modifiable.*



Static IP address: 192.168.1.41
Subnet Mask: 255.255.255.0
Default Gateway: 192.168.1.0
MAC Address: 00-04-a3-00-00-00
DNS Primary Address: 0.0.0.0
DNS Secondary Address: 0.0.0.0
DHCP: ☐ Enable

Wireless Configuration
Authentication: None
Encryption: None
Radio: ☐ 802.11a ☐ 802.11b ☐ 802.11g ☐ 802.11n
Encrypt Config Data: ☐
Pass phrase:
Network Key: 64-bit
ReKey Internal:
SSID:

```
Connecting to COM2

TinyBooter v4.0.16
CirrusEP9302_BSP4_
SJJ Build V4.0.3.4
ARM Compiler versi
Pinging... TinyBoo
User Cancel...
```

Fig 2.20 - Set Static TCP/IP

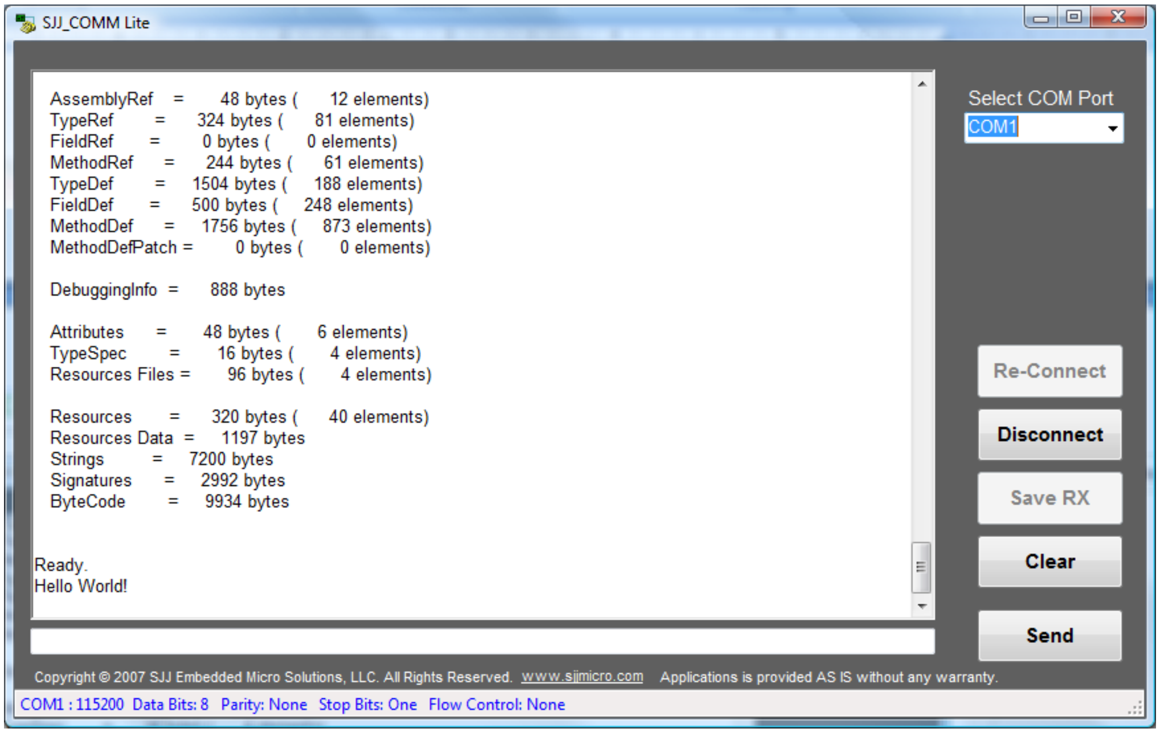
12. Click the Update button when finished.
13. The network configuration settings will be set and the CLR will load after the boot delay time.

Note: *to set the static IP address via the CLR, you do not have to disconnect MFDeploy. Simply boot TinyBooter, allow it to load the CLR and then select* ***Target->Configuration->Network*** *and follow the same steps as 10. & 11. Above. After the configuration has been updated, the CLR will reboot after the MFDeploy boot delay time.*

2.5 SJJ_COMM Lite Utility

Although MFDeploy supports a COM port – debug connection, the utility in this mode is not totally passive and can cause some issues on the debug port. Windows Vista and Windows 7 don't provide HyperTerminal for terminal communication. To address these two issues, the SJJ_COMM Lite utility provides basic ASCII serial communications for capturing debug output. SJJ_COMM Lite requires .NET Framework 2.0. SJJ_COMM Lite is preconfigured for the correct COM port parameters (115200, 8-N-1) to talk to the iPac-9302. All you need to do is select the correct COM port. When you select the COM port from the drop-down, it automatically connects to that COM port. When you want to download from Visual Studio, click the **Disconnect** button and SJJ_COMM Lite will disconnect from the COM port. When you are ready to monitor, again, simply click the **Re-Connect** button.

The monitor window will show ASCII data as it receives it on the selected COM port and will scroll as required, If you wish to save the data monitored to a file, simply disconnect and click the **Save RX** button and you will be presented with a save dialog box. To clear the monitor window, click the **Clear** button.



```
AssemblyRef    =      48 bytes (      12 elements)
TypeRef        =     324 bytes (      81 elements)
FieldRef       =       0 bytes (       0 elements)
MethodRef      =     244 bytes (      61 elements)
TypeDef        =    1504 bytes (     188 elements)
FieldDef       =     500 bytes (     248 elements)
MethodDef      =    1756 bytes (     873 elements)
MethodDefPatch =       0 bytes (       0 elements)

DebuggingInfo  =     888 bytes

Attributes     =      48 bytes (       6 elements)
TypeSpec       =      16 bytes (       4 elements)
Resources Files =     96 bytes (       4 elements)

Resources      =     320 bytes (      40 elements)
Resources Data =    1197 bytes
Strings        =    7200 bytes
Signatures     =    2992 bytes
ByteCode       =    9934 bytes

Ready.
Hello World!
```

Copyright © 2007 SJJ Embedded Micro Solutions, LLC. All Rights Reserved. www.sjjmicro.com Applications is provided AS IS without any warranty.

COM1 : 115200 Data Bits: 8 Parity: None Stop Bits: One Flow Control: None

Fig 2.21 - Data Capture

If you monitor the bootup of the system via SJJ_COMM Lite or other terminal application, the beginning of the captured output will appear similar to the following:

```
TinyBooter v4.0.1681.0
CirrusEP9302_BSP4_1 Build Date: Dec 11 2009 15:56:09
SJJ Build V4.0.3.3.b3r
ARM Compiler version 300651
MSdbgV1__*** nXIP Program found at 0x60040000
Estimated Network RAM usage: 129667 (bytes)
ip address from interface info: 0.0.0.0
mac addrress from interface info: 0.50.c2.e.a.f7
MULTICAST SOCKET_ERROR: 10057
SJJ Build V4.0.3.3.b3r
.NetMF v4.0.1681.0
CirrusEP9302_BSP4_1, Build Date:
    Dec 11 2009 15:58:08
ARM Compiler version 300651

TinyCLR (Build 4.0.1681.0)

Starting...
Created EE.
Started Hardware.
MSdbgV1MSdbgV1?o?i????_MSdbgV1??#~?????%?y????_???8v?????RO^????MSdbgV1MSdbgV1MSdb
gV1?_yN????_???I????_??M_????MSdbgV1MSdbgV1_?b!!????MSdbgV1_?G:)????MSdbgV1_?
9????MSdbgV1<???????0_??????MSdbgV1MSdbgV1_
2???????e?????MSdbgV1__]??????Z_J?????MSdbgV1___NP|?????H???????MSdbgV1MSdbgV1__~H
??????x?=?????????????uw?????MSdbgV1___?m????????-
??????_????????_?????MSdbgV1MSdbgV1????????MSdbgV1MSdbgV1MSdbgV1MSdbgV1MSdbgV1MSdb
```

```
gV1MSdbgV1MSdbgV1MSdbgV1MSdbgV1MSdbgV1MSdbgV1MSdbgV1?_??????__MSdbgV1MSdbgV1MSdbgV1MSdbgV
1MSdbgV1g?(_????uJ?_????s?__????_W??.????ER_9????C??>????_MSdbgV1MSdbgV1)???????_M
SdbgV1
{??????_MSdbgV1O_I?????_MSdbgV1s?_?????_MSdbgV1MSdbgV1_MSdbgV1??
y????_MSdbgV1?QLY????_?>aA????_MSdbgV1_MSdbgV1_MSdbgV1?+_6????__MSdbgV1?hq????_MSd
bgV1_t_?_????_MSdbgV1f?__????_H??!????_ZtG6????MSdbgV1MSdbgV1????????_MSdbgV1MSdbg
V1????????????????????????_??[?????MSdbgV1_MSdbgV1_MSdbgV1_$???????_.???????t?:???
??_~?!?????_H#`??????kei????MSdbgV1MSdbgV1_MSdbgV1_?[
_????MSdbgV1_?
    ?9????%??f????MSdbgV1_7[_q????MSdbgV1__,?V????MSdbgV1_C!C_????MSdbgV1_Q??
    ????MSdbgV1_?iF?????????????????????#p??????A??????MSdbgV1MSdbgV1__?|3??????6
_????????????????i????MSdbgV1_???_????MSdbgV1??R!??????d1????MSdbgV1____MSdbgV1MSd
bgV1MSdbgV1MSdbgV1?8???????jE??????
s???????????????q?????MSdbgV1?_????????G??????__?????????????:??i????6_.f????"X_v?
???____MSdbgV1l_?.????`?_!????_????????MSdbgV1MSdbgV1_MSdbgV1MSdbgV1MSdbgV1MSdbgV1
OW?p????C?| ????_MSdbgV1=??_????_MSdbgV1_MSdbgV1??/ ????_?5nG????_MSdbgV1???7????j
"(?????_P_??????_No debugger!
Create TS.
 Loading start at 6004008c, end 6004eed0
Attaching file.
Assembly: mscorlib (4.0.1681.0) (3572 RAM - 29700 ROM - 17609 METADATA)
```

What looks to be garbage characters debug frames for MFDeploy and can be ignored.

2.6 Upgrading the .NET MF on the iPac-9302

When updates to the bootloader and CLR are made available, the Cirrus download.exe utility is used to update TinyBooter and the MAC address (if required). MFDeploy will be used to deploy the updated CLR.

There are three parts to updating the .NET Micro Framework on the EDK/EDKPlus. The first is downloading the MAC address (if required), the second is the downloading the bootloader binary, and the third is deploying the CLR binary.

Part 1 – Download the MAC Address – Optional if already installed

Every network port must have a unique MAC address. The MAC address is supplied in a small binary file that must be downloaded before downloading TinyBooter. The MAC address is placed in a reserved region of flash that the Ethernet driver can access. The download.exe “option –o” for load address offset must be used. The Cirrus download.exe utility is supplied in the ZIP file with the exercises.

If you have a board with the MAC address installed, then you can skip this step.

1. Power down the iPac-9302.
2. Make sure the Serial cable is connected between the iPac-9302 and the development machine.
3. Move Jumper JB4 (near the JTAG connector) from normal to serial (pins 1 and 2).

Jumper JB4		Configuration
1	2	Serial Boot through UART 1
2	3	Normal Boot from Flash.

4. Open a command window, and run download.exe in a CMD shell as follows:

   download –p n –o 0x7f8000 mac_XXXXXX.bin

   where: n is the COM port number, i.e. 1 for COM1, 2 for COM2, etc
   where: –o <offset> address.bin

***Note**: download -h will list all command line options*

5. The utility will prompt to power on the iPac-9302. Power up the board.

6. Wait for the update to complete. The download utility will burn the MAC address to flash. Be patient as it goes through the flash write operation. You do not want to interrupt power or press the reset switch during the flashing operation.

7. Leave the command window open to download the bootloader, go to Part 2

Part 2 – Downloading the new TinyBooter
In order to download the bootloader, the Cirrus download.exe utility is used. This utility is supplied in the ZIP file with the exercises. To use the download utility:

1. Power down the iPac-9302.
2. Make sure the Serial cable is connected between the iPac-9302 and the development machine.
3. Move Jumper JB4 (near the JTAG connector) from normal to serial (pins 1 and 2).

Jumper JB4		Configuration
1	2	Serial Boot through UART 1
2	3	Normal Boot from Flash.

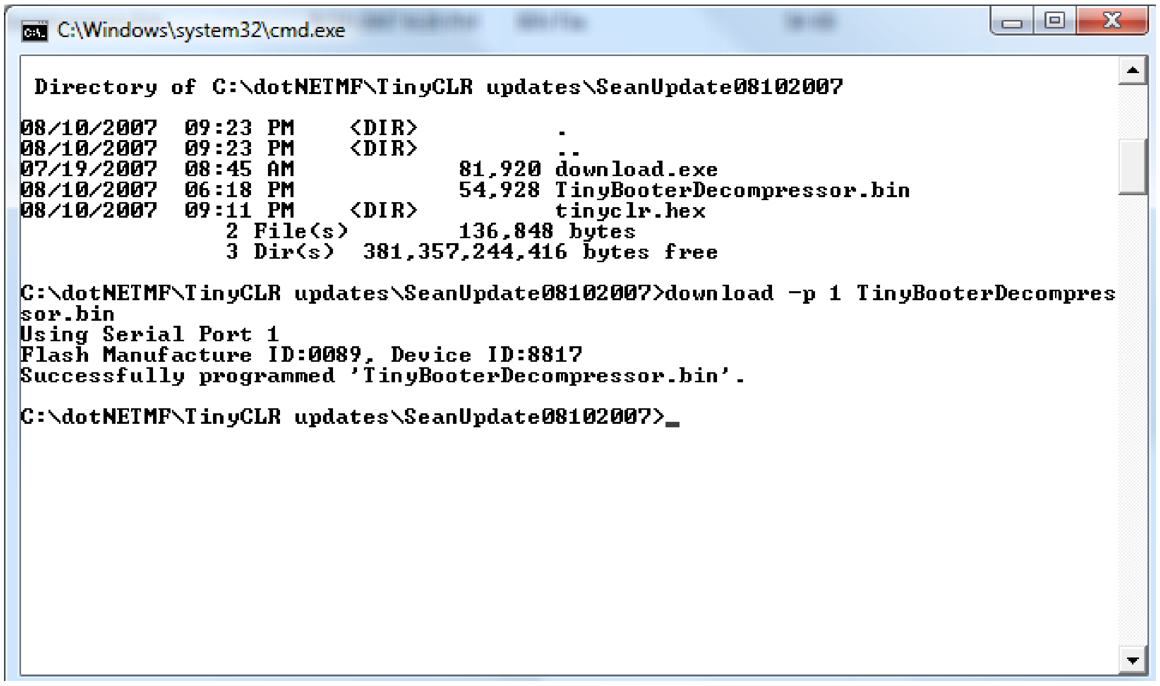
4. In the command window enter the following to download the bootloader:

```
download -p n TinyBooterDecompressor.bin
```

where: n is the COM port number, i.e. 1 for COM1, 2 for COM2, etc

***Note**: download -h will list all command line options*

5. The utility will prompt to power on the iPac-9302, reset the iPac-9302.



```
 Directory of C:\dotNETMF\TinyCLR updates\SeanUpdate08102007

08/10/2007  09:23 PM    <DIR>          .
08/10/2007  09:23 PM    <DIR>          ..
07/19/2007  08:45 AM            81,920 download.exe
08/10/2007  06:18 PM            54,928 TinyBooterDecompressor.bin
08/10/2007  09:11 PM    <DIR>          tinyclr.hex
               2 File(s)        136,848 bytes
               3 Dir(s)  381,357,244,416 bytes free

C:\dotNETMF\TinyCLR updates\SeanUpdate08102007>download -p 1 TinyBooterDecompres
sor.bin
Using Serial Port 1
Flash Manufacture ID:0089, Device ID:8817
Successfully programmed 'TinyBooterDecompressor.bin'.

C:\dotNETMF\TinyCLR updates\SeanUpdate08102007>_
```

Fig 2.22 - Downloading TinyBooter

6. Wait for the update to complete. The download utility will first download the new bootloader image into RAM, and then it will burn that image to flash. Be patient as it goes through the flash write operation. You do not want to interrupt power or press the reset switch during the flashing operation.
7. Power down the board.
8. Once completed, replace JB4 jumper to the normal (pins 2 and 3) boot position. The system is ready for the next step.

Part 3 – Deploying the new CLR image
The final step is to deploy the new CLR image.

1. Open MFDeploy
2. Set "Device" to serial and the serial port that is going to be used for download.
3. Make sure the Serial cable is connected between the iPac-9302 and the development machine.
4. Select **Target->Connect**.
5. Click the Image File Browse button to browse to the new CLR HEX binary files.

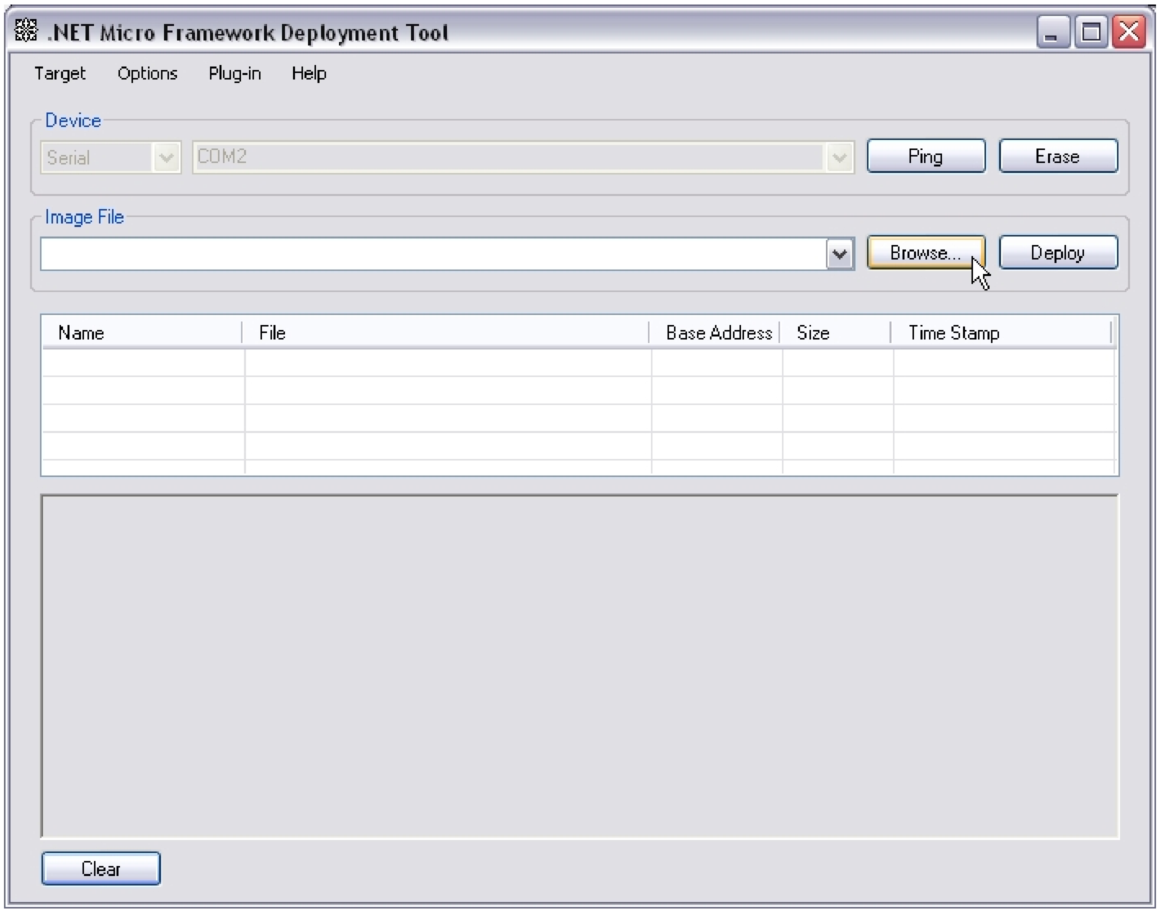


Fig 2.23 - MFDeploy CLR Image File Browse

6. Navigate the browse dialog and select the 2 CLR HEX binary files. Select both files.

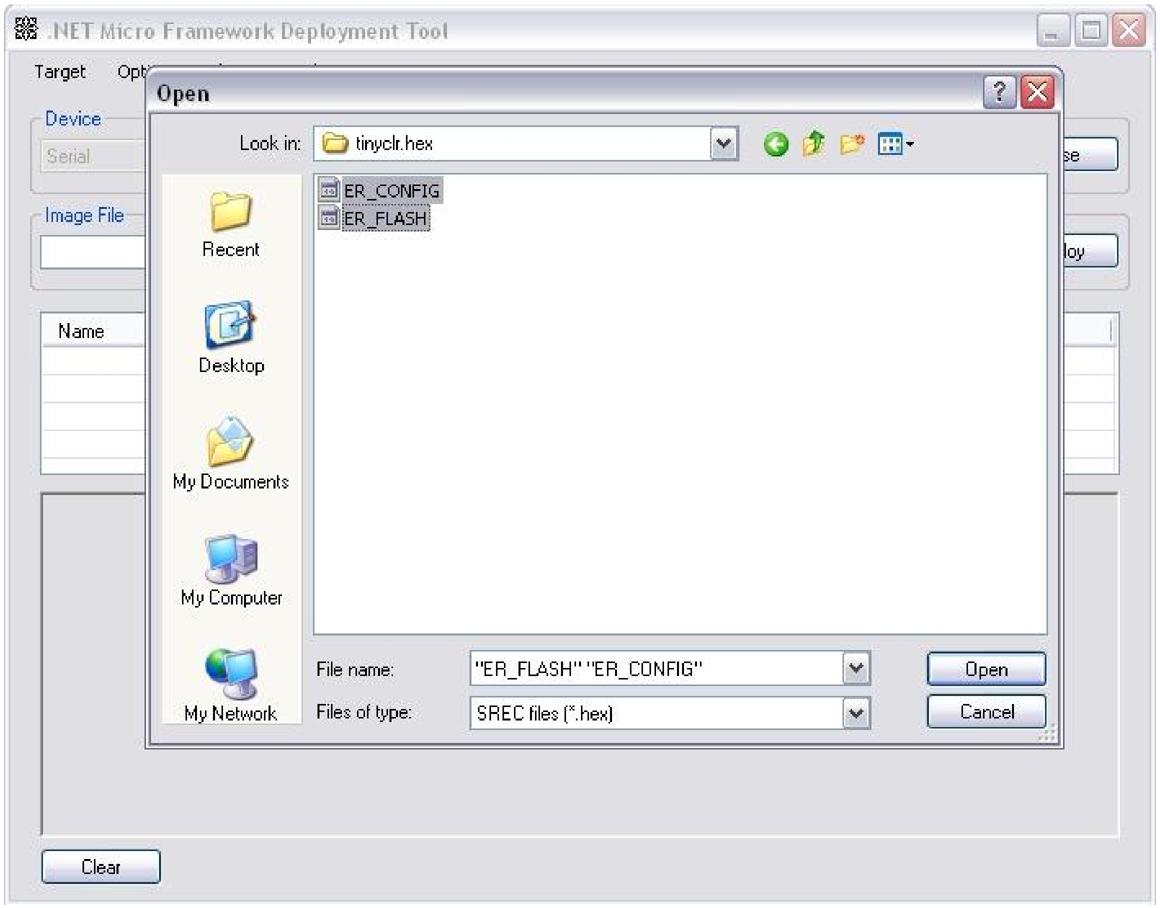
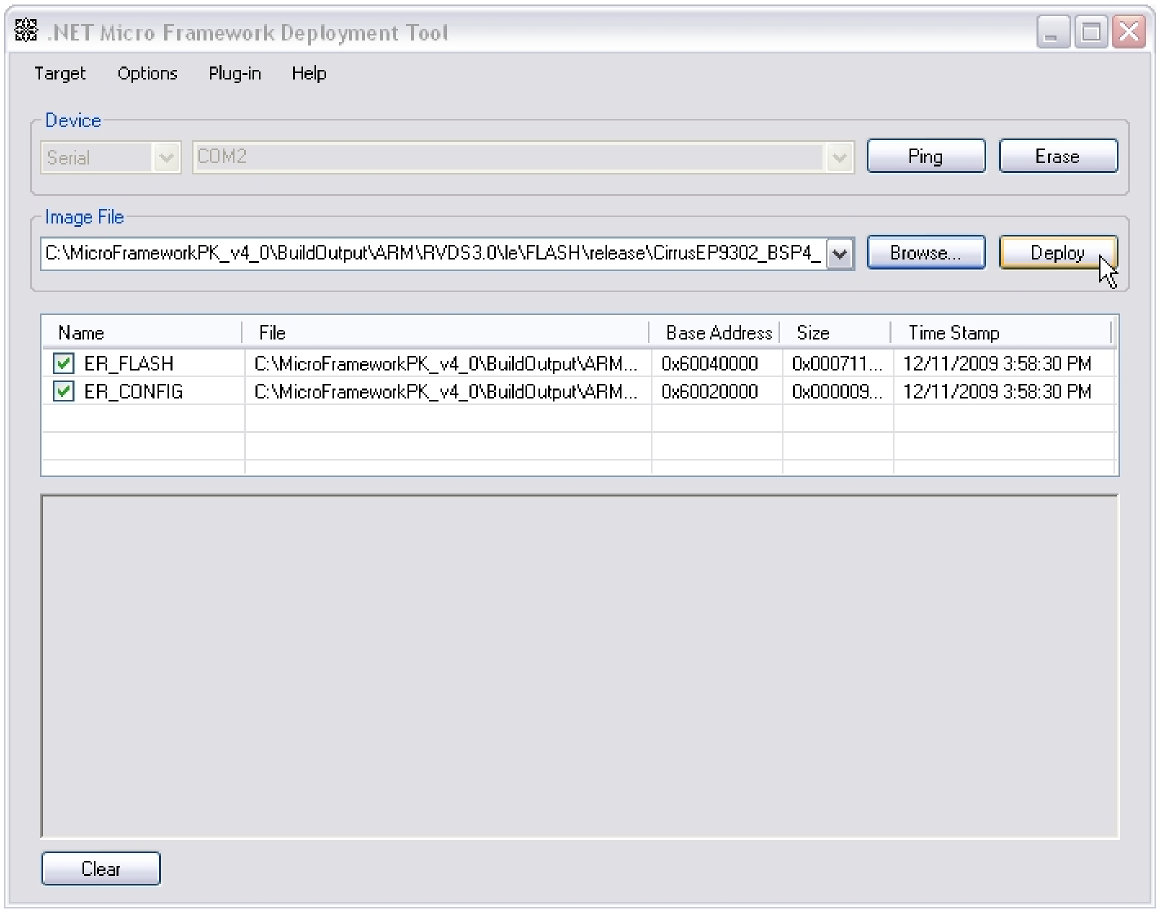


Fig 2.24 - MFDeploy Select CLR HEX Binary Files

7. Power on the iPac-9302 board.
8. When TinyBooter boots up signs on, click on **Ping** to verify that MFDeploy is talking to TinyBooter.
9. Select **Target->Disconnect**.
10. Click on Image File **Deploy**.



Device: Serial, COM2

Image File: C:\MicroFrameworkPK_v4_0\BuildOutput\ARM\RVDS3.0\le\FLASH\release\CirrusEP9302_BSP4_

Name	File	Base Address	Size	Time Stamp
✓ ER_FLASH	C:\MicroFrameworkPK_v4_0\BuildOutput\ARM...	0x60040000	0x000711...	12/11/2009 3:58:30 PM
✓ ER_CONFIG	C:\MicroFrameworkPK_v4_0\BuildOutput\ARM...	0x60020000	0x000009...	12/11/2009 3:58:30 PM

Fig 2.25 - MFDeploy Image File Deploy

11. When finished deploying the new CLR image, select Target Connect.
12. Reboot the iPac-9302 and observe that the target boots to TinyBooter and then loads the new CLR. The system is ready for applications.

2.7 Important note for those upgrading from V2.5 with Ethernet

.NET MF V2.5 release included support for Ethernet. Some registry keys were required in order to get around some timing issues with Visual Studio for application deployment. These registry keys caused longer communications delays and increased the number of retries when Visual Studio was connecting to the target device. These delays are not needed for V4.0. These keys should be removed. The following are the registry keys that must be removed:

```
[HKEY_CURRENT_USER\Software\Microsoft\.NetMicroFramework\NonVersionSpecific\Timing\AnyDevice]
"timeout"=dword:0000ea60
"retries"=dword:0000001E
"override"=dword:00000001
```

2.8 Summary

The chapter covered the features supported with the V4.0 .NET MF on the iPac-9302. Make sure that you have set up the board properly noting the revision of the board. MFDeploy and SJJ_COMM Lite were discussed and will be used with the various exercises. The next chapter dives into the concept of .NET MF and your first .NET MF application

3 The .NET Framework Concept and the first .NET Micro Framework Application

Many programming languages support object oriented programming (OOP). Most of these languages use the same concepts and principles to support reusable code. Here, we are going to look at the common definitions and then dive into the .NET MF SDK to write a program.

3.1 OOP Put into Practice

In OOP, the key word is “Object”. When we are reusing code, we are taking advantage of objects that have been already created or objects we create ourselves. An “Object” is a little program snippet that manipulates and communicates information and manage data. Objects can be as simple as a math equation or represent something as big as a network connection. They contain what are called “Properties” and “Methods”. Basically, Properties represent data and Methods represent actions. The Properties of a particular Object are data values that describe the attributes of the thing the Object represents. The Methods of a particular Object are the actions (subroutines or functions) that manipulate whatever the object represents.

The classic example of an object is to think of a file on a hard drive. A file can have properties such as date created, size, name, author, etc. You can perform actions on a file such as move, create, delete, read, write, save, etc. You can have a file object that provides the properties and methods to act on any file, and you can re-use this code many times in an application.

In Visual Studio, you will hear and read about classes. A “Class” is an object’s definition. Classes and objects are interchangeable in the world of Windows programming. Many of the I/O objects that will be accessed by your .NET MF managed code application will be based on classes defined by the .NET MF class libraries.

The other word you will run across is “Instance”. When you use the class of an object in a program, you are getting an Instance of the class. A distinction is made that you are actually getting a copy of the class and not the actual class itself. You can perform all sorts of manipulations with the copy, but the original class will not change. This is at the heart of OOP.

Many classes can share the same methods and properties, and thus can be grouped into a base class. Where the base class holds the common methods and properties, and the individual classes hold the methods and properties unique to the sub-class.

For example, a Cessna and a Boeing 747 are airplanes, but have slightly different features.

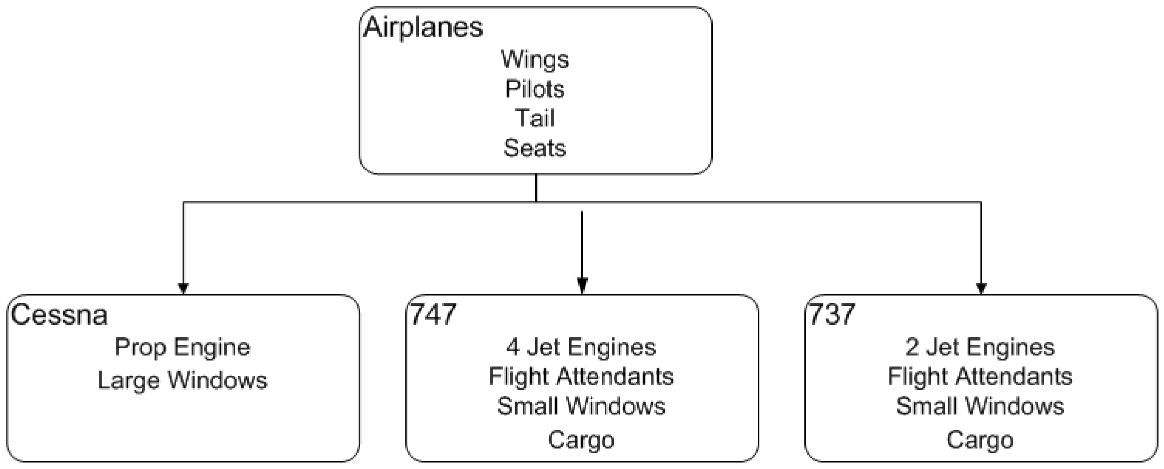


Fig 3.1 - Airplane Class Example

Another example is categories in a book store. The common feature is they are books, but books of the same type are stored in a common location: fiction, Non-fiction, biography, history, travel, computer, etc. The categories make it easier to locate a specific book.

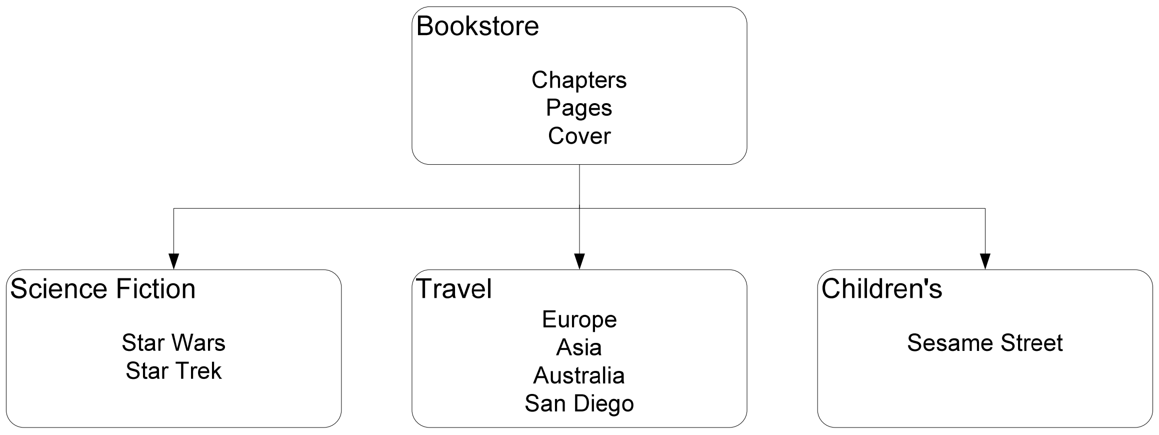


Fig 3.2 - Bookstore Class Example

The subclasses "inherit" properties and methods from the base class. Inheritance is an importance part of OOP.

When we write .NET MF programs, we will take advantage of what is called "namespaces". Namespaces use "dot" syntax to provide a hierarchy scheme for the various classes. This makes programming a little easier and the editor can help out as you type in your code.

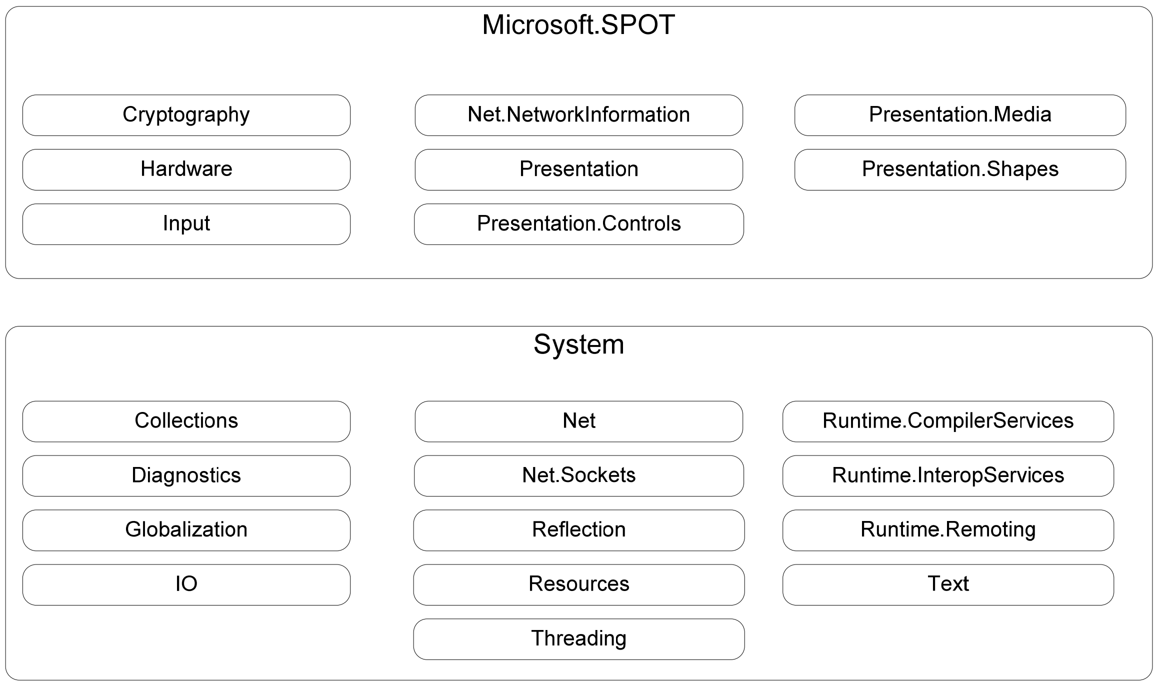


Fig 3.3 - .NET MF Namespaces

3.2 C#, Jitting, and Garbage Collection

C# is currently the only programming language supported by .NET MF. C# is a powerful, component-oriented language. C# is at the heart of the development of .NET Framework. If you are used to writing applications in C, C++, or Java, then the transition to C# should not be difficult.

Unlike C or C++ of the past, when C# code is compiled, it is turned into an executable module called an assembly. An Assembly has two parts: Intermediate language (IL) which contains the executable portion of the program, and Metadata, which describes the assembly's contents.

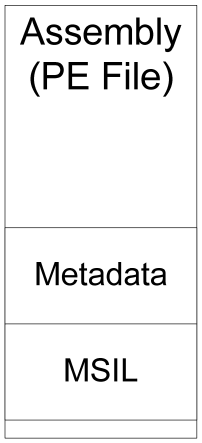


Fig 3.4 - Assembly

When the application is executed, the .NET Framework runtime compiles the assembly into actual machine code for execution. It will not compile the complete code all at once, only the portions it needs to execute. This is called Just-In-Time compilation or "jitting" or JIT. When we compile the .NET MF code, we will create intermediate language assemblies that will be downloaded to the CLR for execution using the CLR's JIT compiler.

Besides compiling on the fly, there is another activity that is going on behind the scenes. Memory management in an embedded system is one of the most critical features for a system's durability. The biggest programming task with a real-time OS is to manage memory in the application, and many programs don't free up the memory for reuse. Un-freed memory (aka memory leaks) impacts the performance of the system and can shorten the uptime of a system. .NET and Java were created to help with this issue specifically by implementation of automatic memory management or "garbage collection". .NET MF includes just such a memory manager. As object instances get destroyed, the garbage collector runs from time to time to free up memory for the system.

3.3 .NET MF SDK

The .NET MF SDK contains assembly files, documentation, example applications, and tools. You need to install Visual Studio 2008 with SP1 or higher, or Visual C# Express 2008 before installing the .NET MF SDK. Visual Studio 2010 is not yet supported. To be sure that you have the latest service pack installed, after you complete the installation of Visual Studio, launch it and from the drop-down click on "Help" and then "Check for Updates". This will run Microsoft Update and will check for service packs or any other updates. If Visual Studio 2008 Service Pack 1 (SP1) has not been installed with your base installation, it will be indicated when you check for updates. Go ahead and install SP1 and then close Visual Studio for now.

Note: *If Microsoft Update does not find Visual Studio 2008 SP1, it can be downloaded from: http://www.microsoft.com/downloads/details.aspx?displaylang=en&FamilyID=fbee1648-7106-44a7-9649-6d9f6d58056e*

The SDK gets installed in the "\Program Files\Microsoft .NET Micro Framework" directory. Version numbers are used to name folders for different versions of the assembly files. This is to help support future versions of the .NET MF. The different subdirectories are as follows:

Sub Directory	Notes
Assemblies	These are the .NET MF class libraries used to build applications
Documentation	Help files
Fonts	Fonts for use with SideShow projects
Tools	Debugging and emulation support

Table 3.1 - .NET MF V2.0 Sub Directories

The example .NET MF projects can be found in the MyDocuments (XP) or Documents (Vista/7) folder under *Microsoft .NET Micro Framework*. The sample projects demonstrate the different capabilities found in .NET MF.

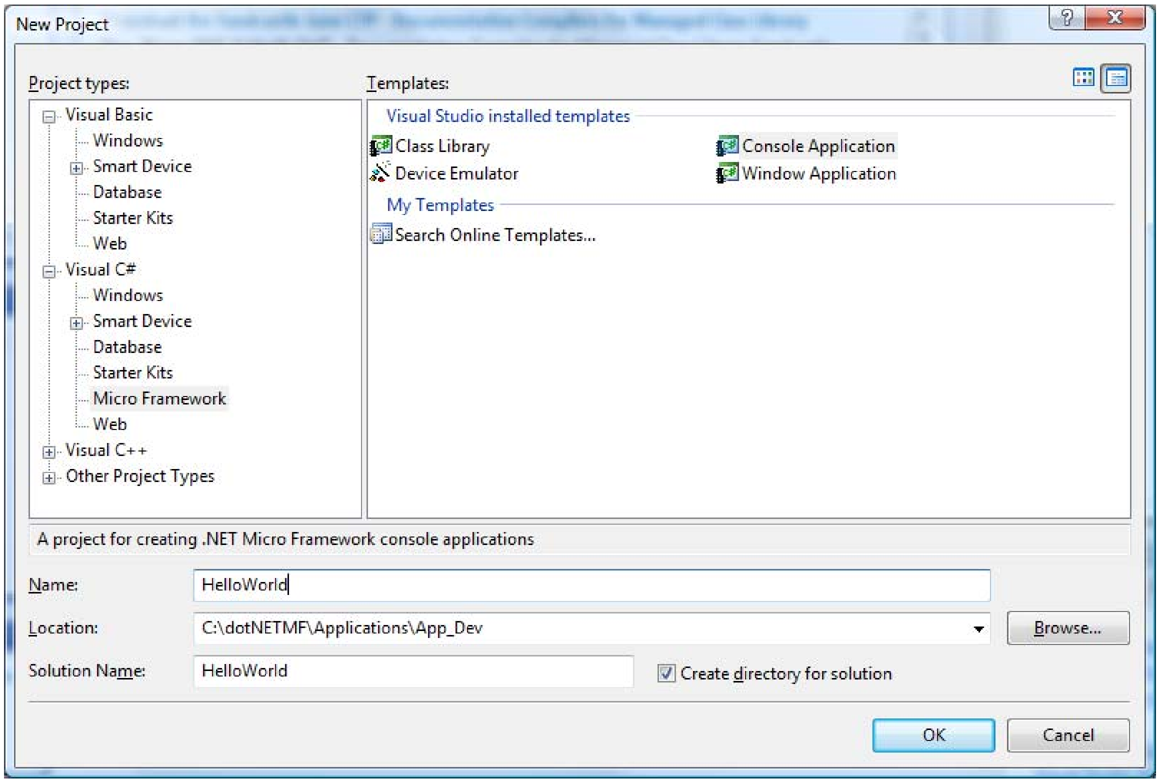
When the SDK is installed, Visual Studio templates are added so you can quickly create classes and applications, which brings us to the first exercise.

3.4 Exercise 3.1: Hello World

Hello World is always a popular getting started application. There is no brain surgery going on in the Hello World application. Its intent is to have something programmatically simple to show you the sequence of basic operations that are needed to create, build, download and run a .NET MF managed code application. Here we will walk through creating a .NET MF application, build it, download it and run it on the iPac-9302 device

Make sure that you have installed Visual Studio and the .NET MF SDK V4.0.

1. Create a folder location on your development PC that you can use to place all exercises. i.e. c:\NETMF_Apps
2. Open Visual Studio or Visual C# Express Edition.
3. From the menu, select **File**->**New**->**Project…**
4. The New Project dialog appears; expand the Project Types on the left side under **Other Languages->Visual C#**.
5. Click on **Micro Framework.**
6. Click on the **Console Application** template.
7. In the Name at the bottom, type in "**HelloWorld**" without the quotes**.**


New Project

Project types:
- Visual Basic
  - Windows
  - Smart Device
  - Database
  - Starter Kits
  - Web
- Visual C#
  - Windows
  - Smart Device
  - Database
  - Starter Kits
  - Micro Framework
  - Web
- Visual C++
- Other Project Types

Templates:

Visual Studio installed templates
- Class Library
- Console Application
- Device Emulator
- Window Application

My Templates
- Search Online Templates...

A project for creating .NET Micro Framework console applications

Name: HelloWorld

Location: C:\dotNETMF\Applications\App_Dev

Solution Name: HelloWorld

[✓] Create directory for solution

Fig 3.5 - Visual Studio New Project Dialog

8. Keep the defaults, and click on the **OK** button.
9. The IDE will create a basic C# application structure. From Solution Explorer in the upper right corner, open the Program.cs file. The program looks like this:

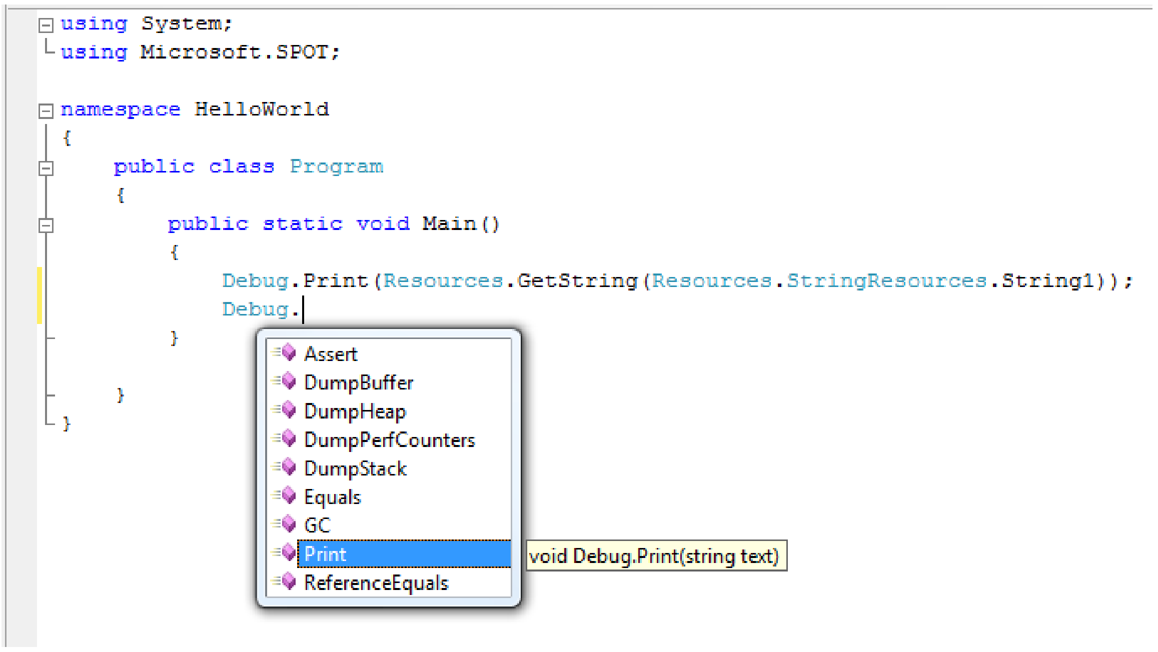
```
using System;
using Microsoft.SPOT;

namespace HelloWorld
{
    public class Program
    {
        public static void Main()
        {
            Debug.Print( Resources.GetString(Resources.StringResources.String1));
        }

    }
}
```

The application structure looks like a regular C# .NET console application, but without the input arguments. The "Using" directives at the top of the code help to simplify using classes and methods within the main body of our application. The .NET Framework contains a number of namespaces with multiple classes. The Using directive removes writing out the whole namespace, class, and/or method. We will take advantage of this directive in our application as we will use the different classes for the iPac-9302.

Everything is contained in the namespace: HelloWorld. Namespaces help to group elements like classes and other name spaces together, and help avoid name collisions with other classes and methods. A basic class is created in the namespace: HelloWorld class. The only method within this class is Main. As you would expect, the Main method is called by the Tiny CLR upon boot to run your application. If you are not familiar with C#, please take a look at Appendix A for a C# Primer and there is a good reference in the bibliography to the book *Inside C# 2nd Edition.*

10. After the first Debug.Print line in Main, add two Debug.Print lines to the code, that say "**Hello World!!!!!**" and "**SJJ welcomes you to .NET Micro Framework development**.".

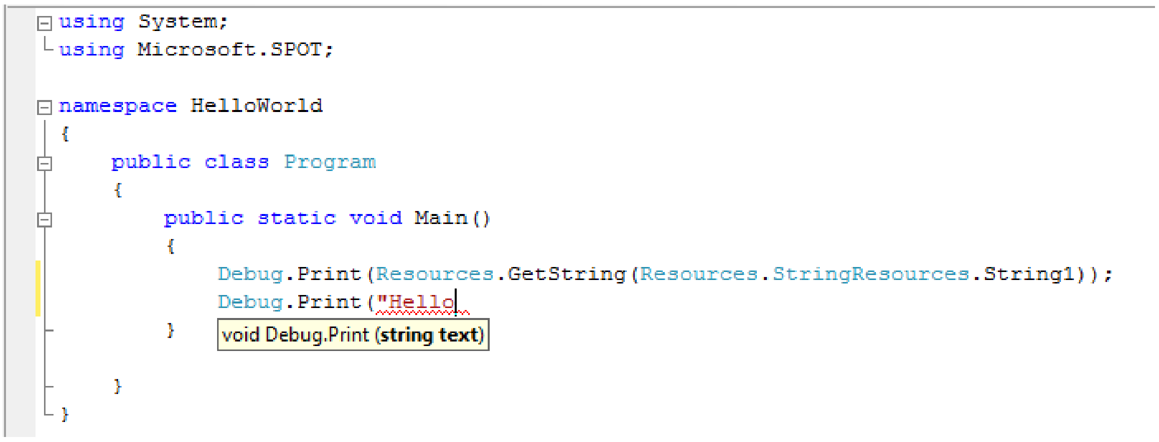
C# is case sensitive; you should start typing with a capital 'D' as you enter "Debug". As soon as you type the first letter, the Intellisense in Visual Studio pops-up with some options. This helps you quickly fill in the code. Once you finish the word Debug and you hit the period '.', the autosense pops up with options specific for the Debug class. You can select Print from the drop down, or if Print is already selected just click the TAB key and the Print will be placed.



```
using System;
using Microsoft.SPOT;

namespace HelloWorld
{
    public class Program
    {
        public static void Main()
        {
            Debug.Print(Resources.GetString(Resources.StringResources.String1));
            Debug.
        }
    }
}
```

Fig 3.6 - Intellisense adding Print method

Once the Print method is typed in, the auto-help will display what is to be passed to the method. In this case it is a string of text. Fill in the first debug string with "**Hello World!!!!!"** and repeat the process for the second string, "**SJJ welcomes you to .NET Micro Framework development**.".



```
using System;
using Microsoft.SPOT;

namespace HelloWorld
{
    public class Program
    {
        public static void Main()
        {
            Debug.Print(Resources.GetString(Resources.StringResources.String1));
            Debug.Print("Hello
        }
    }
}
```

Fig 3.7 - Completing the method call

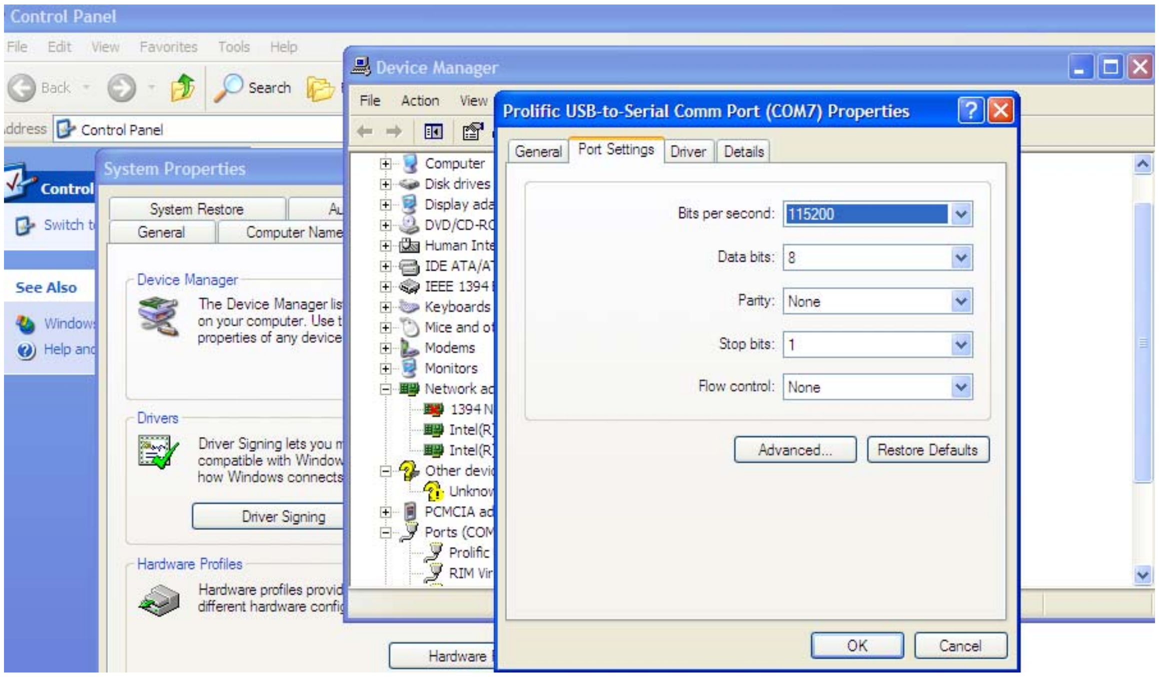
The code should look like the following:

```
using System;
using Microsoft.SPOT;

namespace HelloWorld
{
    public class Program
    {
        public static void Main()
        {
            Debug.Print(Resources.GetString(Resources.StringResources.String1));
            Debug.Print("Hello World!!!!!!");
            Debug.Print("SJJ welcomes you to .NET Micro Framework development!");

        }

    }
}
```

Earlier we talked about objects and inheritance. The reason we can just type “Debug” is because the Microsoft.SPOT name space is defined in the using directive. Otherwise you would have to type out the full method call: Microsoft.SPOT.Debug.Print. This gets a little long and tedious. The *using* directives with the different name spaces help us get to the class quickly, so we can concentrate on creating the application rather than on typing.

11. Save the project.
12. Now let’s try running the Hello World application on the iPac-9302.
13. Make sure the Null modem cable is connected between the iPac-9302 and the development computer.
14. Using Device Manager, make sure that your COM port is set to the following:

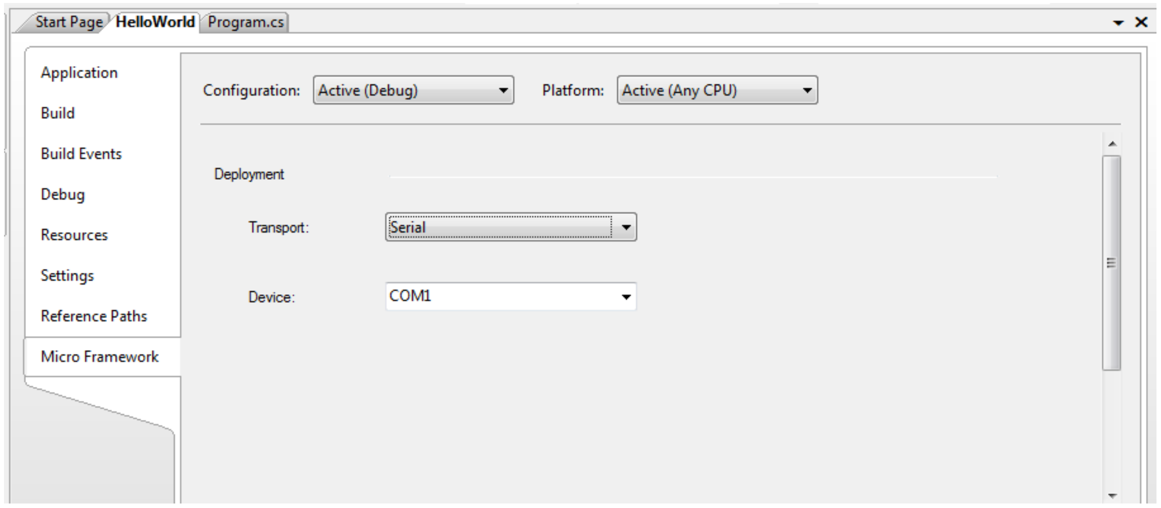
- Baud: 115200
- Data Bits:8*
- Parity: None
- Stop Bits: 1
- Hardware Flow control: None



Bits per second: 115200
Data bits: 8
Parity: None
Stop bits: 1
Flow control: None

Fig 3.8 - COM Port Settings

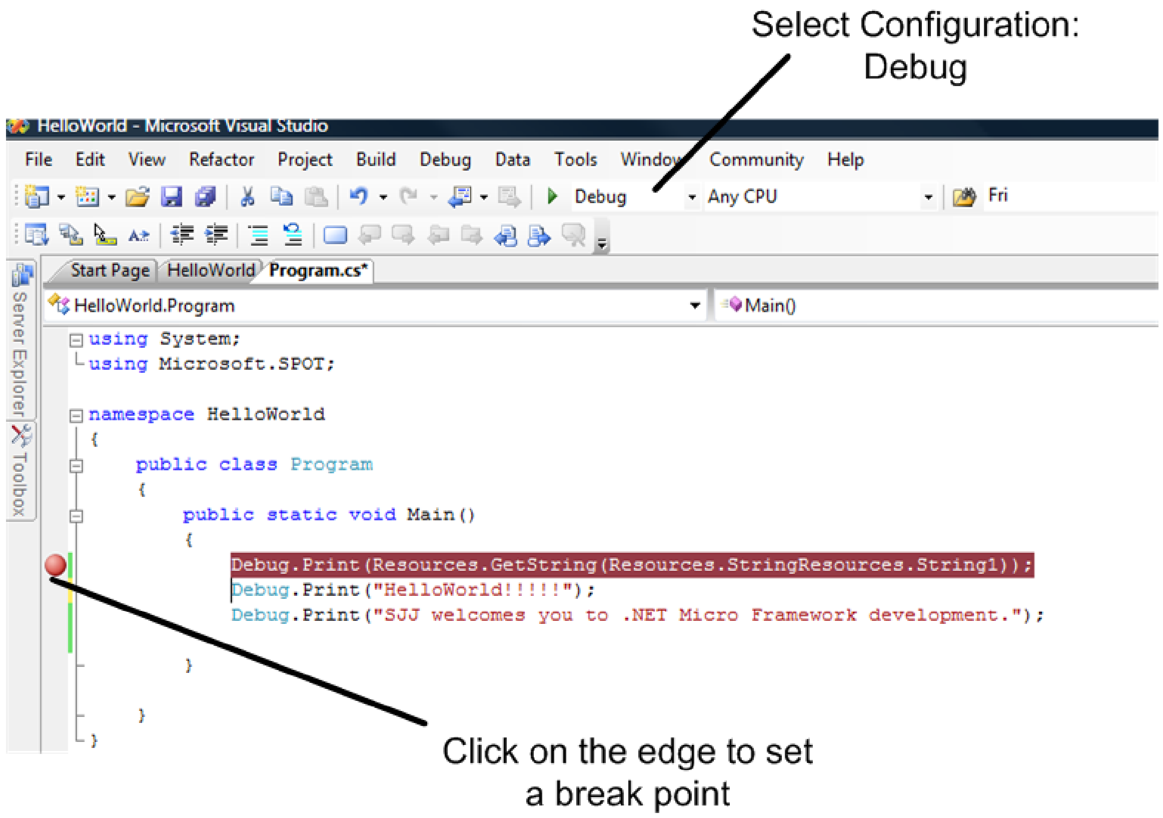
15. From the menu, with the HelloWorld project selected in the Solution Explorer, select **Project** and then select **HelloWorld Properties…** from the submenu.
16. The project properties page appears, click on the **Micro Framework** tab.
17. Under **Deployment**, select **Serial** for the Transport and **Com#** as the Device. Where # is the COM port number of the development computer to which you have connected the null modem cable.



Configuration: Active (Debug)
Platform: Active (Any CPU)
Transport: Serial
Device: COM1

Fig 3.9 – Hello World Properties: COM Port Selection

18. Save the project.
19. Now, make sure Debug is select for the configuration.
20. Set a break point on the first Debug print line.


Select Configuration: Debug
HelloWorld - Microsoft Visual Studio
File Edit View Refactor Project Build Debug Data Tools Window Community Help
Debug
Any CPU
Start Page
HelloWorld
Program.cs*
HelloWorld.Program
Main()
Server Explorer
Toolbox
Click on the edge to set a break point

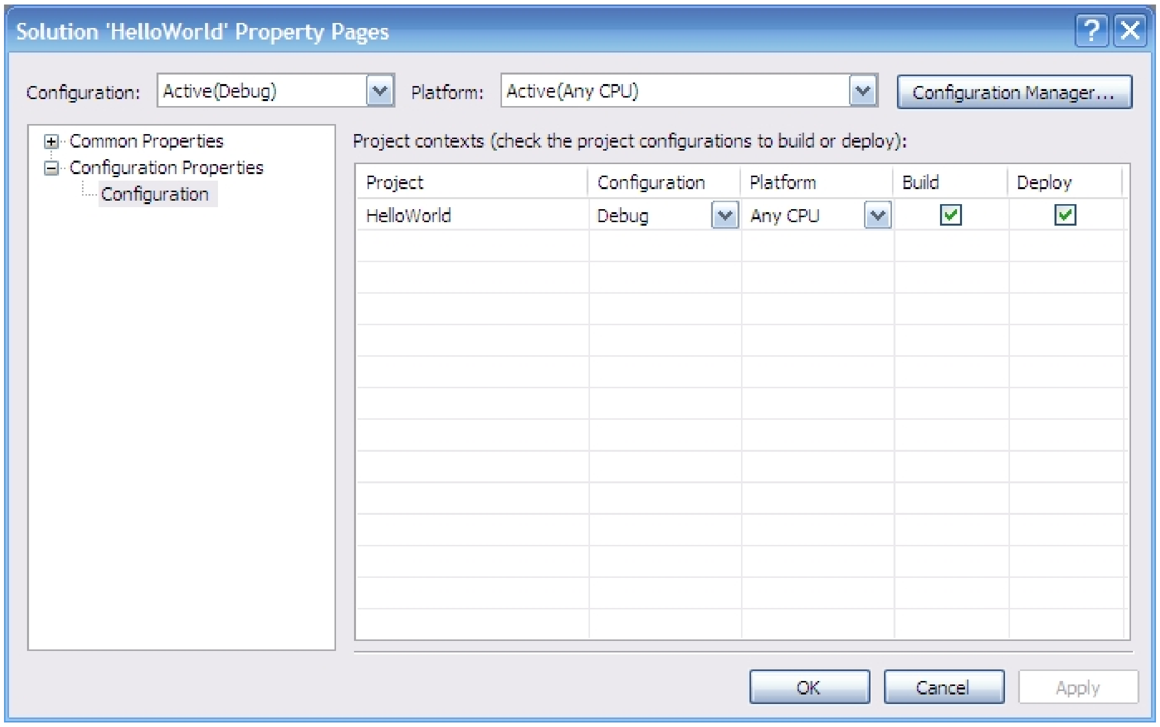
```
using System;
using Microsoft.SPOT;

namespace HelloWorld
{
    public class Program
    {
        public static void Main()
        {
            Debug.Print(Resources.GetString(Resources.StringResources.String1));
            Debug.Print("HelloWorld!!!!!");
            Debug.Print("SJJ welcomes you to .NET Micro Framework development.");

        }

    }
}
```

Fig 3.10 - HelloWorld Build & Debug Configuration

21. From the menu, select **Build->Build Solution**. The application should build without errors. If there are errors, make the appropriate corrections, and rebuild the image.
22. Power up the iPac-9302. You need to wait at least 20 seconds before deploying an application. This gives TinyBooter a chance to turn control over to the CLR. You can also monitor the boot-up of the iPac-9302 with MFDeploy and use the clearing of the BootLoader Flag to shorten the boot time, but **remember to disconnect MFDeploy before you attempt to deploy an application from Visual Studio.** Only one application can have the COM port open at a time. If MFDeploy is connected when you attempt an application deployment, the deployment will fail.
23. From the Visual Studio menu, select **Build->Deploy HelloWorld**. This will download the application to theiPac-9302.

When an application is downloaded to the hardware, the current application is stopped, the application flash memory area is erased, and the new application is burned to flash.

If the deployment fails, check the COM port settings under properties and try again. Also check for any application you may have running that uses the serial port you have connected for deployment, like MFDeploy or a terminal emulator program. Also check the Solution Properties Pages. In the Solution Explorer either select the Solution 'HelloWorld' and from the dropdown select **Project->Properties**; or right-click the Solution 'HelloWorld' and select **Properties**. In the left pane, expand **Configuration Properties** and click **Configuration**. Make sure that the Deploy checkbox is selected for both the Debug and Release configurations.



Configuration: Active(Debug) Platform: Active(Any CPU)

Project contexts (check the project configurations to build or deploy):

Project	Configuration	Platform	Build	Deploy
HelloWorld	Debug	Any CPU	✓	✓

Fig 3.11 - Solution Property Pages

If you still cannot deploy your solution, you might also have previously deployed a bad application that is interfering with the download or an application that has the CPU so busy, that the CLR cannot respond to the connect request frames being sent by Visual Studio. You may have to use MFDeploy to erase the application area as we described in section 2.4.2, *Clearing the previous application*.

24. To start the debug session, hit the F5 key or from the menu select **Debug->Start Debugging**. The application will be downloaded and the Visual Studio debugger window opened. The application will run to the breakpoint.
25. From the menu, select **View** and then **Output** from the drop down.
26. You can use the familiar debug step-over, step-in, etc. to walk through the code. As you step through the code, you will see the print statements appear in the output window.
27. Stop the debugger and close the solution when finished.

You can use SJJ_COMM to view the output or you could also receive the debug output from a terminal application like Tera Term or HyperTerminal. Set the COM port to the following:

- Baud: **115200**
- Data bits: **8**
- Parity: **None**
- Stop bits: **1**

Note: *Windows Vista no longer supports HyperTerminal. SJJ_COMM can be used to capture debug output. Make sure that you disconnect SJJ_COMM, HyperTerminal, or any other terminal program when doing downloads from Visual Studio.*

3.5 Debug Class and Download applications

We just used the Debug class to generate an output message. This may not seem like a big deal, but the class has other methods as well:

Method	Description
Assert*	Overloaded. Causes a break in execution if the specified assertion (condition) evaluates to false.
GC	Runs garbage collection, a service that automatically reclaims unused computer memory.
Print	Prints a message (followed by the current line terminator) to the standard output stream.

Fig 3.12 - Debug Class

* Not available for the EDK

The different Debug methods can be used to help output performance information to help with debugging an application. The first method is available when an instrumented kernel is built into the port. The iPac-9302 doesn't have the instrumented kernel installed.

The iPac-9302 uses the serial port to download applications. A conflict can occur where a Debug.Print call will interfere with a download of a new application. Care must be made to avoid situations where a Debug.Print call is constantly being made which would prevent a new application from downloading, ie, Debug.Print in an infinite loop. A condition like this may require you to erase the application from the device before you can download an update build or different application.

3.6 Deployment Issues

You will run into issues when deploying applications to the hardware. The problems can vary from two Windows applications trying access the same COM port to a managed code application deployed to the device that is crashing CLR. Here are some typical problems and solutions:

- Application download fails – first try to download again. Then check the project properties settings and the solution properties.
- Another application is using the COM port. Visual Studio needs exclusive access to the COM port, so make sure no other application is connected to the same COM port, like your terminal emulator program or MFDeploy.
- Periodic output to the debug port from a previously deployed managed code application will cause downloads to fail. Debug.Print called in an infinite loop will prevent applications from being downloaded. The best solution is to use MFDeploy to Erase the application space and then try the download, again.
- Application fails to download no matter what - Toggle the Transport selection in the Project Properties back to **Emulator**, click **save** while in the Project Properties window, force a complete rebuild by selecting from the drop-down **Build->Rebuild Solution**, and then click **save**. Go back to the Transport selection in the Project Properties and reselect **Serial** and select the appropriate COM port, click **save** while in the Project Properties window, force a complete rebuild by selecting from the drop-down **Build->Rebuild Solution**, and then click **save**. Then try to download, again. Visual Studio sometimes gets stuck and forgets where to deploy the applications. If this fails to remedy the download failure, close Visual Studio, open visual studio, again, and go through the transport exercise, again.

Note: *Clearing out the application space is always a good way to make sure that the download will work.*

3.7 Summary: OOP to running on the Hardware

We cover the basics of going from an OOP application to running it on the device in this chapter. Object Oriented Programming is a major development to help write applications from reusable code. .NET Framework and .NET Micro Framework technologies are based on this concept. There are many books that dive deeper into the theories and operation of OOP, the CLR, and C#, but the basics were provided here as a groundwork for the following chapters.

The HelloWorld example demonstrated the Debug class and the Print method. It also demonstrated configuring a Visual Studio solution for download and debug. Best of all we are able to download the application to the iPac-9302 and run it.

4 Inner Process Mechanics

Writing .NET MF applications is very similar to writing C# applications for Windows. As we mentioned earlier, you can take some examples that run on the desktop and run them in .NET MF. The real embedded development involves architecting the applications. Careful architecture can save time and money in the long run. In this section, we will cover the following CLR-kernel details:

- Threads
- Thread priorities
- Interrupts

4.1 The Inner Processing Mechanics - Threads, Priorities, Round-Robin, and Interrupts.

Traditionally, when learning a new operating system or embedded kernel, the details of the kernel's capabilities are discussed. We needed to cover some of the basic application and hardware features first before we dived into this topic. Since the CLR is a managed code kernel, many of the typical real-time requirements in other operating systems and kernels are built in. if you are familiar with threads in .NET Framework, it is the same in .NET Micro Framework. Here is what the CLR supports:

- Threading – A thread is a single unit of execution. Threads allow an application to separate different operations into individual threads. Threading support in .NET MF is the same support as in .NET Framework where the application is running as a thread that can spawn other threads.

- Priority levels – Just like .NET Framework, there are 5 priority levels: Highest, Above Normal, Normal, Below Normal, and Lowest. The application and threads all start with the Normal priority level.

- Round Robin – Threads at the same priority run in a round robin of 20 milliseconds (ms) each, but every thread will get processed at some point.

- Interrupts - One very important item to understand is that hardware interrupts in .NET MF will NOT pre-empt the system. For a queued interrupt to run, either the current task or thread must sleep, wait, or the 20 mSec system timer times out to switch from the current thread to other pending threads. Interrupt latency can be as high as 20 mSec or slightly more depending on the runtime dynamics of the CLR at the time of the interrupt. This is very important when it comes to architecting the application; especially when high-frequency interrupts are coming in.

The application itself is run as a thread. Creating other threads is used to divide the work of the application. All threads are queued to run in a loop with Highest priority threads being queued first and lowest priority threads being queued last. After the system 20 mSec time slice, a sleep, or a wait, the CLR checks to see if a hardware interrupt has come in. If a hardware interrupt has been queued, the CLR switches out of the current thread, and it will schedule the interrupt before running any other threads. If no interrupts are queued, the CLR switches out of the current thread and switches in the next thread in the queue. Exercise 4.1 will demonstrate this concept, but first let's talk about creating threads.

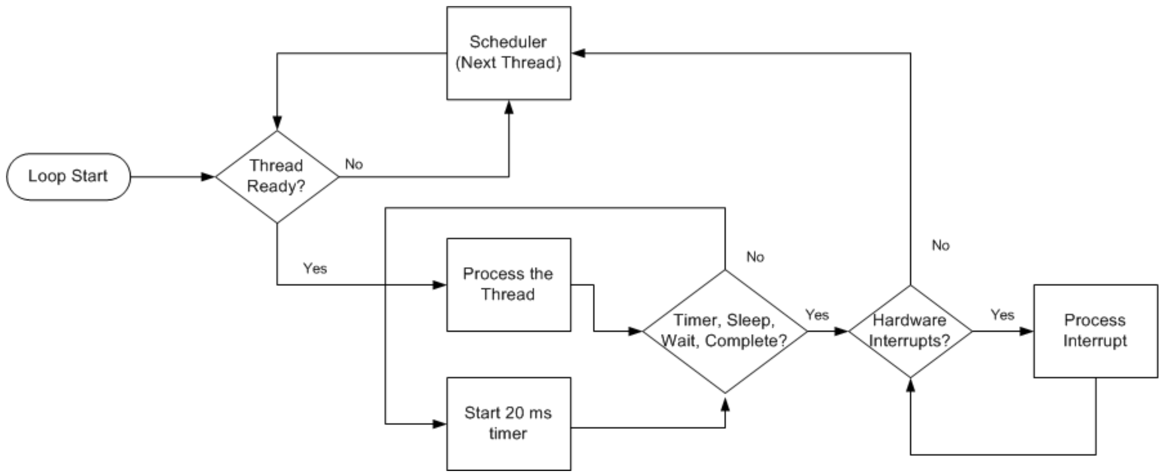


Fig 4.1 - CLR Thread Scheduling

4.1.1 Creating Threads

Threads are simply methods that perform some action. The methods can be static, if the method is only accessed in the same class or non-static if the method is to be accessed by a different class. Non-static methods allow for access to instance variables.

1. The first step is to create the methods that will perform the action.
2. Define a Thread method using the ThreadStart delegate. For example:

```
ThreadStart work = new ThreadStart(myMethod);
```

3. Next, define the new thread

```
Thread workerThread = new Thread(work);
```

4. Once defined, you can then start the thread:

```
workerThread.Startr();
```

A common short hand notation for creating the thread would be to combine steps 2 and 3 into one line:

```
Thread workerThread = new Thread(new ThreadStart(myMethod));
```

Once the thread has been defined, you can get the state of the thread with the ThreadState enumeration members:

Thread State	Description
Aborted	Thread is stop because of an abort request
AbortRequested	Another thread is requesting the thread to stop
Background	Thread is running as a background thread
Running	Thread is active
Stopped	Thread is inactive
StopRequested	The thread has a request to stop
Suspended	The thread is suspended
SuspenedRequested	Another thread is requesting the thread to be suspended
Unstarted	The thread has yet to be started
WaitSleepJoin	Thread is blocked waiting for another object to be signaled, is a sleep, or is wait to join another thread once it completes processing.

Fig 4.2 - ThreadState Enumeration Members

4.2 *Exercise 4.1: Looking at Processing Mechanics in Action*

In this example we are going to look at the BasicProcessing project that has already been created in the Chapter 4 folder of the examples. The application creates 4 threads on 4 methods that perform pretty much the same actions:

```
string myString;
for (int y = 0; y < 50; y++)
{

    myString = "Thread 1 " + y.ToString();
    //Debug.Print(myString);
    //Thread.Sleep(10);

}
Debug.Print("[Thread 1] completed");
```

There is also an Interrupt port defined for FGPIO1. The GreenLED will alternate on or off with each interrupt that comes in. Two exercises in the next chapter shows how to use a function generator or the 555 timer circuit to generate interrupts at a rate of 30Hz or lower. Also notice that there is a Debug.Print statement in the method processing the FGPIO1 interrupt. Typically, you want your interrupt callbacks to be as short as possible. Debug.Print is too long and complex of a process, but for demonstrating how things are running in the process loop, it makes for a handy, though not recommended example.

A terminal communication program is required to capture the debug output. You can use any terminal application, but the instructions will refer to the SJJ_COMM Lite program.

Note: *If you open the project that is supplied with the EDK, due to possible directory differences, you will have to remove and re-add the SJJ_HardwareProvider resource.*

Note*: This lab assumes that you are familiar with downloading and running applications on the iPac-9302.*

1. Open Visual Studio or Visual C# Express Edition.
2. Open the BasicProcessing Project in the Chapter 6 folder of the EDK examples.
3. Since we will be downloading to the iPac-9302, you will need to setup the COM port for your system. With the project selected in the Solution Explorer, from the Project menu drop down, select **Project->BasicProcessing properties…**
4. In the Micro Framework tab, set the Transport to **Serial** and the Device to the COM port on your development machine.

5. Build and download the application to the iPac-9302.
6. Start SJJ_COMM, and select the COM port to connect to. We will not be using interrupts at this time, so don't apply the FGPIO1 interrupt.
7. Power cycle the iPac-9302. You will see the debug output for the Tinybooter and CLR scroll by. Then the application runs and outputs the following:

```
Hello World!
[Thread 1] completed
[Thread 2] completed
[Thread 3] completed
[Thread 4] completed
[Main] completed
Done.
```

Each thread completed in the order it was started since all threads are at the same priority level.

8. In SJJ_COMM, disconnect from the COM port
9. As a test, uncomment the Debug.Print lines in each of the 4 methods and the Main for-loops.
10. Build and download the application again.
11. Using SJJ_COMM Lite, connect to the COM port.
12. Power cycle the iPac-9302. The application outputs each of the values of the for-loops. You can see that each thread is processed some time and then the next thread runs. The main is also scheduled to run as well. To provide easier viewing, the start of the output has been put into columns.

```
Hello World!          Thread 1 8          Thread 3 5
Main 0                Thread 1 9          Thread 3 6
Main 1                Thread 2 0          Thread 3 7
Main 2                Thread 2 1          Thread 3 8
Main 3                Thread 2 2          Thread 3 9
Main 4                Thread 2 3          Thread 4 0
Main 5                Thread 2 4          Thread 4 1
Main 6                Thread 2 5          Thread 4 2
Main 7                Thread 2 6          Thread 4 3
Thread 1 0            Thread 2 7          Thread 4 4
Thread 1 1            Thread 2 8          Thread 4 5
Thread 1 2            Thread 2 9          Thread 4 6
Thread 1 3            Thread 3 0          Thread 4 7
Thread 1 4            Thread 3 1          Thread 4 8
Thread 1 5            Thread 3 2          Thread 4 9
Thread 1 6            Thread 3 3          Main 8
Thread 1 7            Thread 3 4          Main 9
```

13. In SJJ_COMM, Clear the output and disconnect from the COM port.
14. Now let's look at thread priorities, but first comment out the Debug.Print lines in each of the for-loops.
15. There are 4 commented thread priority lines just after public void Run(). Un-comment, the lines for theThread1.Priority and theThread2.Priority.
16. Build and download the application again.
17. Using SJJ_COMM Lite connect to the COM port.
18. Power cycle the iPac-9302. The output looks like the following:

```
Hello World!
[Thread 2] completed
[Thread 3] completed
[Thread 4] completed
[Main] completed
```

```
[Thread 1] completed
Done.
```

Thread 2 was set to above normal and Thread 1 was set below normal. Thread1 finished last as expected. You cannot tell that Thread 2 was above normal looking at this output, and Thread2 was started first.

19. In SJJ_COMM, Clear the output and disconnect from the COM port.
20. Uncomment theThread3.priority line.
21. Build and download the application again.
22. Using SJJ_COMM Lite connect to the COM port.
23. Power cycle the iPac-9302. The output looks like the following

```
Hello World!
[Thread 3] completed
[Thread 2] completed
[Thread 4] completed
[Main] completed
[Thread 1] completed
Done.
```

Thread3 was set to highest and finishes first.

24. In SJJ_COMM Lite, Clear the output and disconnect from the COM port.
25. Uncomment theThread4.priority line.
26. Build and download the application again.
27. Using SJJ_COMM Lite connect to the COM port.
28. Power cycle the iPac-9302. The output looks like the following

```
Hello World!
[Thread 3] completed
[Thread 2] completed
[Main] completed
[Thread 1] completed
[Thread 4] completed
Done.
```

Thread 4 is set to lowest; and, as expected, it will finish last. You can play with the different priorities and changing the start order of each thread to get a feel for the processing of multiple threads.

29. In SJJ_COMM Lite, Clear the output and disconnect from the COM port.
30. Comment the Debug.Print lines in each of the 4 methods and the Main for-loops.
31. Build and download the application again.
32. Using SJJ_COMM Lite, connect to the COM port.
33. Power cycle the iPac-9302/

```
[Thread 3] completed
[Thread 2] completed
[Thread 1] completed
[Thread 4] completed
Done.
```

4.3 Summary: Architect the Image!

The previous chapter provided the basic information on how to create, download, and debug applications. Here we looked at the inner-process of an application by the CLR. If there is anything that we can stress in developing an embedded system, student project, or hobby project, is that time should be taken to architect the system. A careful plan will lead to a more robust and stable solution.

A good understanding of the inner process mechanics is very important when designing the whole system. You don't want to get locked into only writing an application and forget about the system as a whole. You need to take into account how external devices interact with the system. A device that overruns the system with interrupts can stop other processing from running, which bring us to the next chapter on GPIOs

5 General Purpose Input/Output Pins

The previous chapter introduced the basics to programming .NET MF applications. .NET MF has several classes that control different hardware ports. The most basic of these ports is general purpose input/output pins or GPIOs.

GPIOs are pins that can input or output a logical 0 or 1, and some of these, when configured as inputs, can also be used to trigger interrupts, where code execution can jump to a special interrupt callback handler routine. In this chapter, the following will be covered:

- The different GPIO ports on the iPac-9302.
- The available GPIO classes in .NET MF.
- Exercises that demonstrate accessing the different features of GPIOs.

5.1 *iPac-9302 GPIO*

One of the really nice features of the iPac-9302 is that it comes with GPIO ports, a whole bunch of GPIO ports. Some ports are available directly from the Cirrus EP9302 SOC (System On Chip) and others have been added through the customization of the CPLD. Some of the GPIOs support interrupts when configured at inputs, many support being configured for either input and output operation, while the rest are in dedicated banks of input-only and output-only operation. One of the dedicated output banks provides 8 high drive output GPIOs for connecting to higher current devices like relays.

All of the GPIOs are brought out to different headers on the board for easy access. HDR1 has 3 banks of the dedicated GPIOs from the CPLD, including those configured for higher current drive. HDR 2 has some of the GPIOs that come directly from the Cirrus EP9302 SOC which have alternate functions. This will be discussed in more detail, later. HDR3 has a mixture of GPIOs from the Cirrus EP9302 SOC and from the CPLD.

5.1.1 CPU GPIOs

There are several GPIOs available on the Cirrus EP9302 SOC. The GPIOs are grouped in Ports of 8 pins each. Ports A, B, C, F, E, and H. There are several that are called Extended GPIOs or EGPIOs (Ports A, B, F) that support interrupts. The rest of the GPIOs are used for input / output only. Of course, you can poll these non-interruptible GPIOs when configured as inputs to capture incoming logic 0's or 1's.

The processor doesn't bring out all the pins for each of the ports. 8 pins are available for ports A and B, 1 pin is available on port C, 3 on port F, 2 on port E, and 4 on port H. The iPac uses some of the GPIOs for internal functions, so please review the tables below to see which pins are available for general use.

Many of the CPU GPIOs are multiplexed with an alternative pin functions such as I2S or serial port. These pins are EGPIO4, EGPIO5, EGPIO6, and EGPIO13. In future releases of the .NET MF / iPac-9302 port, these pins maybe switched to the alternative functions, so care should be used when selecting these pins for use with your application.

Port E is a special port for controlling the status LED indicators. The first pin of the port is connected to the green led, which we will use in various exercises. The second pin of the port is connected to the red LED, and is used exclusively by the boot controller for error indication and handling and is not available for general use.

***Note:** outputting to the red LED GPIO pin will cause the iPac to reset and run in an unpredictable fashion.*

On startup, the data registers and the data direction registers for the CPU GPIO pins are cleared. This has the effect of setting all GPIOs to inputs; but left unterminated. the pins will float to a logic 1 or 3.3V output. You will need to take this into account when connecting any circuits to these pins.

There are no internal pullup/pulldown resistors for the internal GPIOs. When asked for resistor mode, always select Disabled. Any other selections will be ignored.

Finally, the SJJ Hardware provider DLL has the correct interrupt mode definitions. There are general definitions pre-defined by the .NET MF assemblies, but you should use the constants from the SJJ Hardware provider DLL. The iPac-9302 does support positive and negative edge triggered interrupts; it only supports one or the other. Level sensitive interrupts are not supported.

5.1.2 CPLD GPIOs

The CPLD GPIOs are hardcoded for either Input or output. The port labeling for these GPIO ports are W, X, Y, and Z. W and Z are input ports. Y and X are output ports. X is a special high current drive port that can supply more current to devices like relays. Jumper JB1 (JB1 and JB5 for RV1) is used to configure the pull up voltage for Port X GPIOs. None of these CPLD GPIOs provide interrupts.

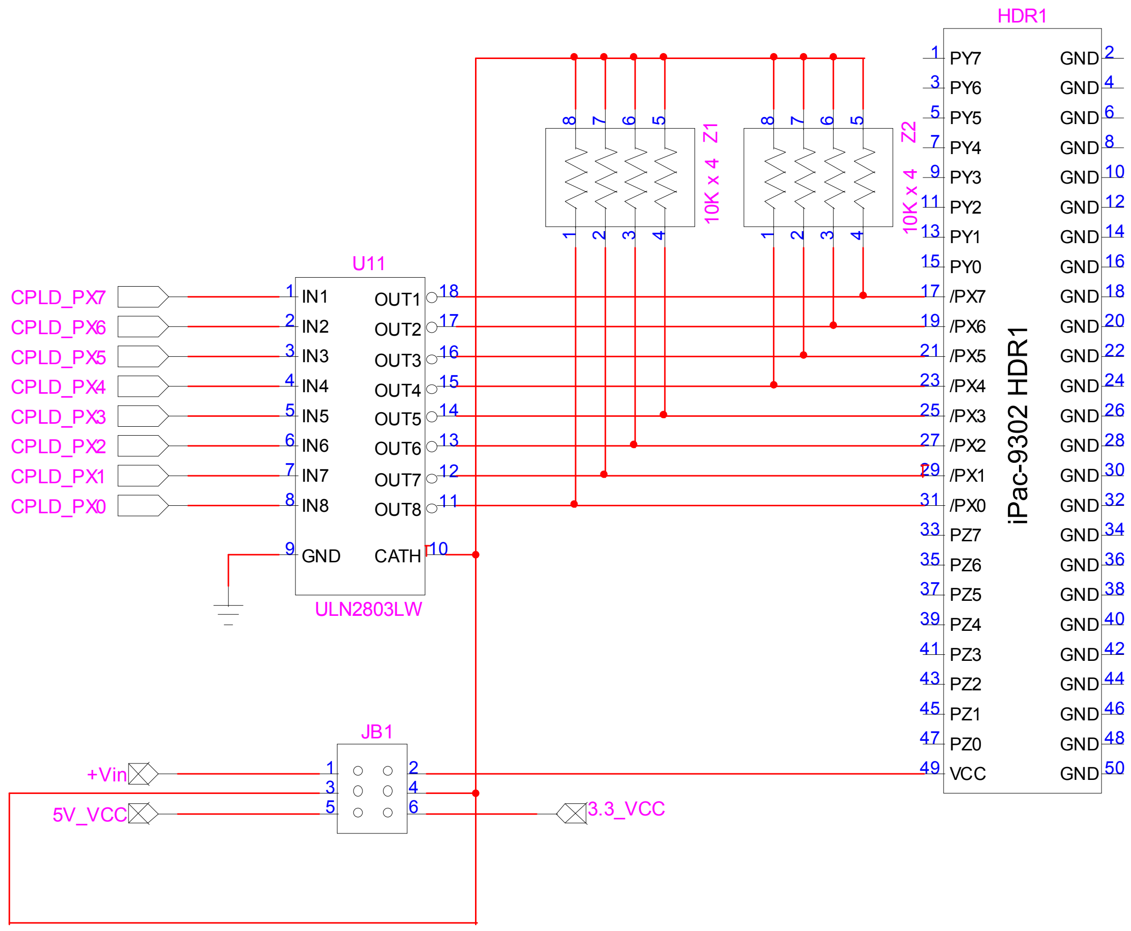


Fig 5.1 - CPLD GPIOs X, Y, & Z for iPac-9302 RV1

5.1.3 iPac-9302 GPIO Summary Table

The following is the GPIO summary table for the iPac-9302 GPIOs that are available for the current port of the .NET MF.

Pin	Signal	Input / Output	Which IC?	Interrupt	Power-On voltage (V)
HDR1-1	PY7	Output	CPLD	No	0[1]
HDR1-3	PY6	Output	CPLD	No	0[1]
HDR1-5	PY5	Output	CPLD	No	0[1]
HDR1-7	PY4	Output	CPLD	No	0[1]
HDR1-9	PY3	Output	CPLD	No	0[1]
HDR1-11	PY2	Output	CPLD	No	0[1]
HDR1-13	PY1	Output	CPLD	No	0[1]
HDR1-15	PY0	Output	CPLD	No	0[1]
HDR1-17	PX7	High Drive Output	CPLD	No	JB1 Volts
HDR1-19	PX6	High Drive Output	CPLD	No	JB1 Volts
HDR1-21	PX5	High Drive Output	CPLD	No	JB1 Volts
HDR1-23	PX4	High Drive Output	CPLD	No	JB1 Volts
HDR1-25	PX3	High Drive Output	CPLD	No	JB1 Volts
HDR1-27	PX2	High Drive Output	CPLD	No	JB1 Volts
HDR1-29	PX1	High Drive Output	CPLD	No	JB1 Volts
HDR1-31	PX0	High Drive Output	CPLD	No	JB1 Volts
HDR1-33	PZ7	Input	CPLD	No	0[1]
HDR1-35	PZ6	Input	CPLD	No	0[1]
HDR1-37	PZ5	Input	CPLD	No	0[1]
HDR1-39	PZ4	Input	CPLD	No	0[1]
HDR1-41	PZ3	Input	CPLD	No	0[1]
HDR1-43	PZ2	Input	CPLD	No	0[1]
HDR1-45	PZ1	Input	CPLD	No	0[1]
HDR1-47	PZ0	Input	CPLD	No	0[1]
HDR2-19	EGPIO[13]/I2S-SDI2	Input/Output	CPU	**Yes**	3.3[2]
HDR2-34	EPGIO[4]/I2S-SDO1	Input/Output	CPU	**Yes**	3.3[2]
HDR2-35	EPGIO[5]/I2S-SDI1	Input/Output	CPU	**Yes**	3.3[2]
HDR2-36	EPGIO[6]/I2S-SDO2	Input/Output	CPU	**Yes**	3.3[2]
HDR3-1	HGPIO[2]	Input/Output	CPU	No	3.3[2]
HDR3-3	HGPIO[3]	Input/Output	CPU	No	3.3[2]
HDR3-5	HGPIO[4]	Input/Output	CPU	No	3.3[2]
HDR3-7	HGPIO[5]	Input/Output	CPU	No	3.3[2]
HDR3-9	FGPIO[1]	Input/Output	CPU	**Yes**	3.3[2]
HDR3-11	CGPIO[0]	Input/Output	CPU	No	3.3[2]
HDR3-13	EGPIO[1]	Input/Output	CPU	**Yes**	3.3[2]
HDR3-15	EGPIO[2]	Input/Output	CPU	**Yes**	3.3[2]
HDR3-17	EGPIO[3]/HDLCLK1	Input/Output	CPU	**Yes**	3.3[2]
HDR3-19	EGPIO[6]/I2S-SDO2	Input/Output	CPU	**Yes**	3.3[2]
HDR3-21	EPIO[10]	Input/Output	CPU	**Yes**	3.3[2]
HDR3-23	EPIO[11]	Input/Output	CPU	**Yes**	3.3[2]
HDR3-25	EGPIO[12]	Input/Output	CPU	**Yes**	3.3[2]
HDR3-27	EGPIO[13]/I2S-SDI2	Input/Output	CPU	**Yes**	3.3[2]
HDR3-31	EGPIO[15]	Input/Output	CPU	**Yes**	3.3[2]

Pin	Signal	Input / Output	Which IC?	Interrupt	Power-On voltage (V)
HDR3-33	PW0	Input	CPLD	No	0[1]
HDR3-35	PW1	Input	CPLD	No	0[1]
HDR3-37	PW2	Input	CPLD	No	0[1]
HDR3-39	PW3	Input	CPLD	No	0[1]
HDR3-41	PW4	Input	CPLD	No	0[1]
HDR3-43	PW5	Input	CPLD	No	0[1]
HDR3-45	PW6	Input	CPLD	No	0[1]
HDR3-47	PW7	Input	CPLD	No	0[1]
Green LED	Green LED	Output	CPU	No	3.3

Table 5.1 - Available iPac-9302 GPIOs for .NET MF

[1] Unterminated CPLD input voltage
[2] Unterminated SOC input voltage

Note: *EGPIO[14] is not available; reserved for future use as PWM.*

The CPLD can be reprogrammed to support other functions than GPIOs. Please contact us for more information.

Warning*: If a GPIO port is incorrectly defined, for example defining a port as Output when it is input only, NO error will be reported back by CLR. The application will fail outright, and you may have to use MFDeploy to erase the application from flash.*

5.2 GPIO Classes, Enumerators, and the SJJ Hardware Provider

The GPIO classes are part of the Microsoft.SPOT.Hardware assembly along with the other hardware IO ports. There are several classes and enumerations that address the GPIO ports.

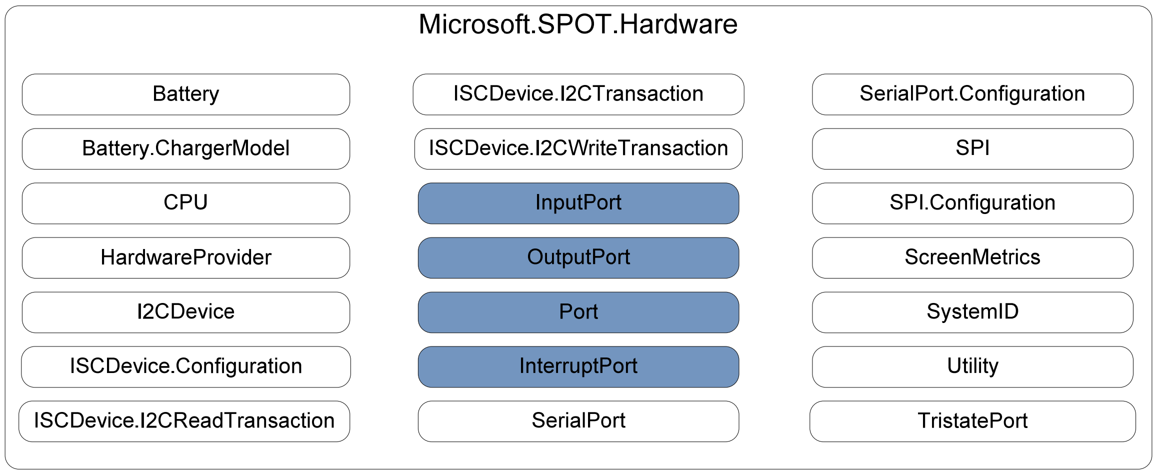


Fig 5.2 - Class members for Microsoft.SPOT.Hardware

5.2.1 GPIO Pin Enumerations

There are three enumerations that are used to help define what is supported in the hardware. The SJJ hardware provider contains these three enumerations to define the GPIO pins, resistor modes, and interrupt modes support by the iPac-9302.

First, there is the Cpu.Pin Enumeration, which defines the GPIO pins available. The GPIO constants in the SJJ hardware provider are named to match the name of the port GPIO and the header pin location on the board.

The Port.InterruptMode Enumeration allows for constants to be provided for the hardware's specific interrupt mode support. The iPac-9302 only supports Positive and Negative edge triggered interrupts.

The Port.ResistorMode Enumeration defines the resistor on the port as being pull up, pull down, or disable. The iPac-9302 doesn't have resistors on the pins, so use only Disable.

5.2.2 GPIO Port Classes

The GPIO Port class are based off a Port Class at the base.

5.2.2.1 Port Class

The Port Class is the base class for managing GPIO pins. The class contains the Port constructor, the port identifier property, and two methods: Dispose and Read. Read is common to the other GPIO classes, and you will find it in each of the 3 new classes.

5.2.2.2 InputPort Class

The InputPort Class consists of the constructor and a couple of properties. The InputPort constructor is used to define pins as input port. There are 3 ways to define the port:

InputPort (Pin *portId*, bool *initialState*, bool *glitchFilter*, ResistorMode *resistor*)
InputPort (Pin *portId*, bool *glitchFilter*, ResistorMode *resistor*)
InputPort (Pin *portId*, bool *glitchFilter*, ResistorMode *resistor*, InterruptMode *interruptMode*)

The InputPort is used to read the state of a GPIO pin, but if you look at the help for the InputPort you will notice that the Read() method is not listed. The reason is that the class inherits the properties of the base class, which is the Port Class.

5.2.2.3 OutputPort Class

The OutputPort Class consists of a couple constructors, the initial state property, and the write method. The Write method is unique to the OutputPort Class. There are two constructors:

OutputPort (Pin *portId*, bool *initialState*)
OutputPort (Pin *portId*, bool *initialState*, bool *glitchFilter*, ResistorMode *resistor*)

The Write method writes a value to the pin – True (logic 1) or False (Logic 0).

5.2.2.4 InterruptPort Class

The InterruptPort Class is used for GPIOs that support interrupt capability. The class consists of a property that gets or sets the interrupt mode, a method that clears the interrupt, and an event that adds or removes callback methods.

The constructor sets up the GPIO pin and the other properties of the interrupt pin.

InterruptPort (Pin portId, bool glitchFilter, ResistorMode resistor, InterruptMode interrupt)

The ClearInterrupt method is used for level sensitive interrupts. Since the iPac-9302 doesn't support these types of interrupts, the method will not be used.

OnInterrupt is used to define the callback from the interrupt. The exercises will demonstrate how this event is setup and used.

The Read method is inherited from the Port class so you can read the state of an InterruptPort.

5.3 *Exercise 5.1: Twiddle a Bit*

The first GPIO application will simply flash the Green Led on and off. The green LED is connected to the processor like so:

Note: *If you open the project that is supplied with the EDK, due to possible directory differences, you will have to remove and re-add the SJJ_HardwareProvider resource.*

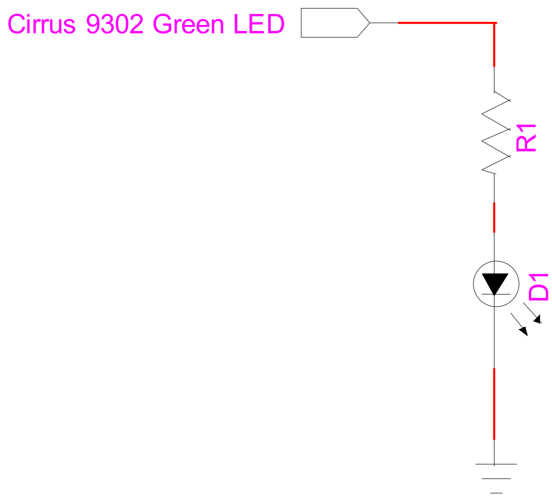


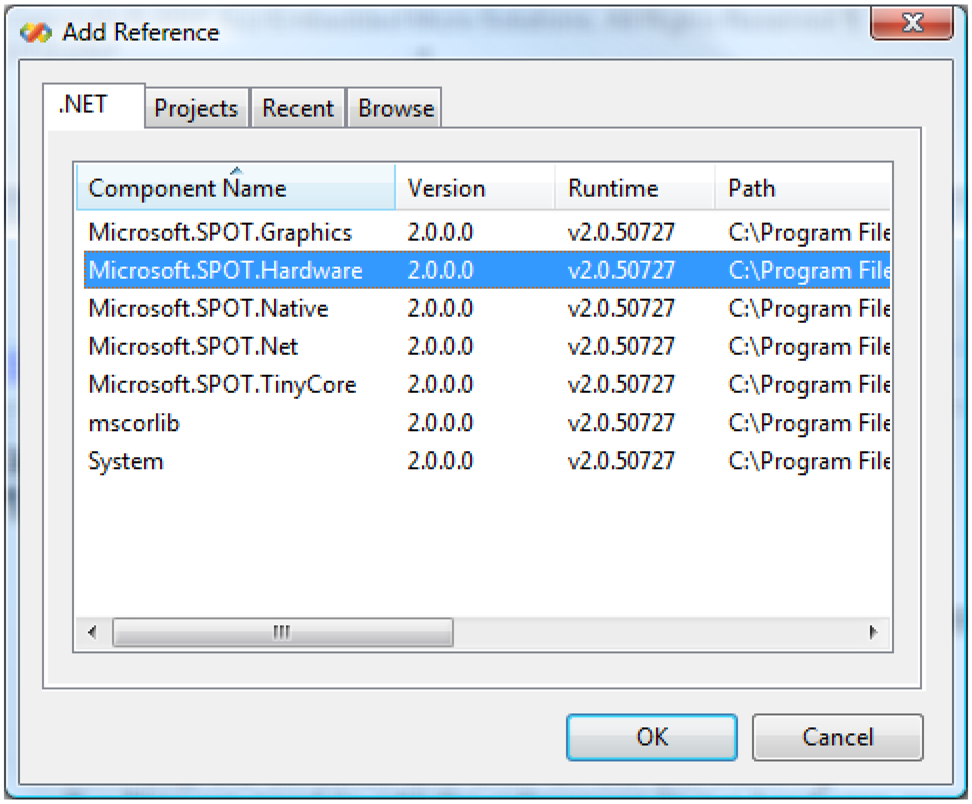
Fig 5.3 - Cirrus EP9302 Green LED

Logic 1 will turn the green LED on, and logic 0 will turn it off.

5.3.1 Create and setup the project

1. Open Visual Studio or Visual C# Express Edition.
2. From the menu, select **File**->**New**->**Project…**
3. The New Project dialog appears; expand the Project Types on the left side under **Other Languages->Visual C#**.
4. Click on **Micro Framework**
5. Click on the **Console Application** template.
6. In the Name at the bottom, type in **GreenLED**
7. Keep the defaults, and click on the **OK** button.

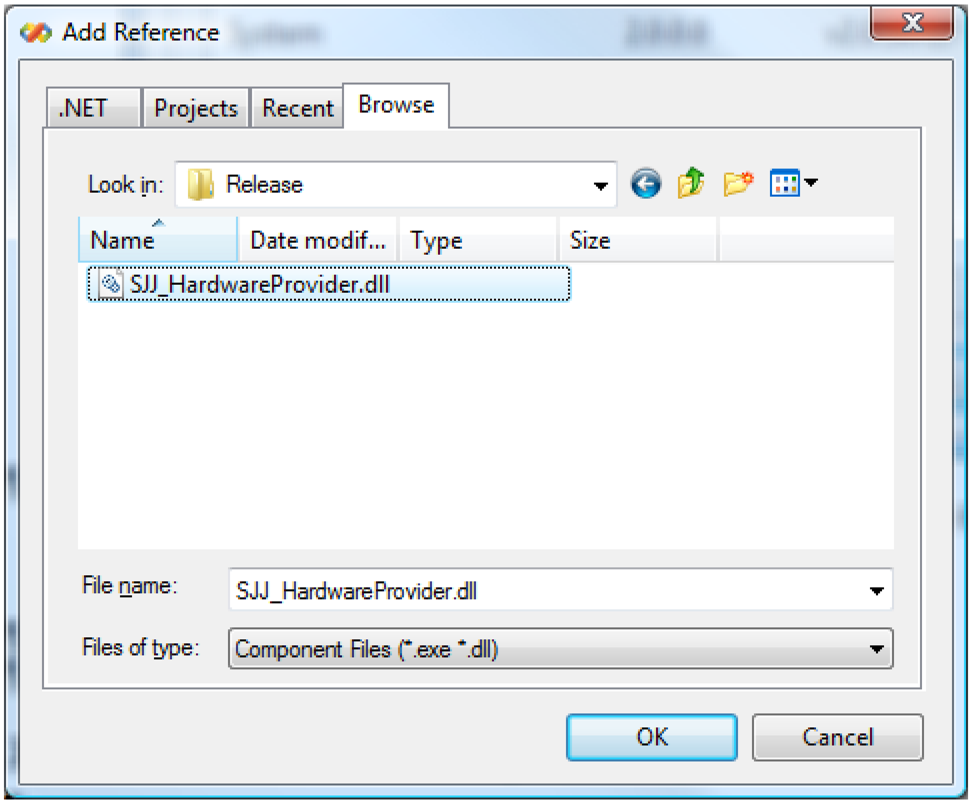
8. Now we need to add the references to our hardware assemblies. With the project selected in the Solution Explorer, from the menu, click on **Project**, and then select **Add Reference…** from the drop down menu items.
9. In the .NET tab of the Add Reference dialog, select **Microsoft.SPOT.Hardware**.



Component Name	Version	Runtime	Path
Microsoft.SPOT.Graphics	2.0.0.0	v2.0.50727	C:\Program File
Microsoft.SPOT.Hardware	2.0.0.0	v2.0.50727	C:\Program File
Microsoft.SPOT.Native	2.0.0.0	v2.0.50727	C:\Program File
Microsoft.SPOT.Net	2.0.0.0	v2.0.50727	C:\Program File
Microsoft.SPOT.TinyCore	2.0.0.0	v2.0.50727	C:\Program File
mscorlib	2.0.0.0	v2.0.50727	C:\Program File
System	2.0.0.0	v2.0.50727	C:\Program File

Fig 5.4 - Visual Studio Add Reference Dialog .NET Tab

10. Click **OK**.
11. From the menu, click on **Project**, and then select **Add Reference…** from the drop down menu items.
12. Click on the **Browse Tab**, locate and select the **SJJ_HardwareProvider.DLL** that came with the exercises.



SJJ_HardwareProvider.dll

File name: SJJ_HardwareProvider.dll

Files of type: Component Files (*.exe *.dll)

Fig 5.5 - Visual Studio Add Reference Dialog Browse Tab

13. Click **OK**. We now have all the references to the classes. In the Solution Explorer, you will see the newly added references under the References folder.
14. Save the project

5.3.2 Adding the code

1. From the Solution Explorer, double-click on **Program.CS** to open the main code module in the editor.
2. Add the following *using* directives to the top of the code listing:

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

3. Now we need to add a new Output port that will be used to access the green LED. After the Debug.Print statement, add the following:

```
OutputPort myGreenLED;
```

We define a reference name of class type OutputPort for the new output port.

***Warning**: If a GPIO port is incorrectly defined, for example, defining a port as Output when it is input-only, NO error will be reported back by CLR. The application will fail outright, and you will have to use MFDeploy to erase the application from flash before you can deploy a new, corrected version.*

4. Now we need to instantiate the new port to the desired port pin. Type in the following:

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

The SJJ_HardwareProvider.dll provides user friendly names for the GPIO Pins. In this case, Pins.GreenLED is the name for the GPIO port (E) with the green LED. As you look through the list of available pins, the pin names correspond to the available GPIOs in the iPac-9302 – see Table 5.1 - Available iPac-9302 GPIOs for .NET MF.

5. Finally, add a while-loop that toggles the Green LED:

```
while (true)
{
    Thread.Sleep(500);
    myGreenLED.Write(true);

    Thread.Sleep(500);
    myGreenLED.Write(false);
}
```

The thread.sleep lets the application put the current thread to sleep for about a half second. The two writes to the port are to turn the green LED on (true), and off (false)

***Warning**: Don't put any Debug.Print statements in the while-loop. This can cause interference with downloading a new application.*

6. Save the project.

The main code module should now look like the following:

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


namespace GreenLED
{
    public class Program
    {
        public static void Main()
        {
            Debug.Print(Resources.GetString(Resources.StringResources.String1));

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

            while (true)
            {
                Thread.Sleep(500);
                myGreenLED.Write(true);

                Thread.Sleep(500);
                myGreenLED.Write(false);

            }

        }

    }
}
```

5.3.3 Configure Project Properties; Build, and Deploy the application.

1. Make sure the null modem cable is connected to the iPac-9302 and the development computer.
2. With the GreenLED project selected in the Solution Explorer, from the menu, select **Project** and then select **GreenLED Properties…** from the submenu.
3. The project properties page appears, click on the **Micro Framework** tab.
4. Under Deployment, select **Serial** for the Transport and **Com#** as the Device. Where # is the COM port number of the development computer to which the null modem cable is connected.
5. Save the project
6. From the menu, select **Build**, and then select **Build GreenLED** from the drop down menu.
7. Make sure the iPac-9302 is powered on. You need to wait at least 20 seconds before attempting to deploy an application. You can also monitor the boot-up of the iPac-9302 with MFDeploy and use the clearing of the BootLoader Flag to shorten the boot time, but **remember to disconnect MFDeploy before you attempt to deploy an application from Visual Studio.**
8. From the menu, select **Build**, and then select **Deploy GreenLED**.
9. The output window should show a successful download. Power down the board, wait one second, and power up the board.

After about 6 seconds, the onboard green LED should start flashing on and off about once a second.

5.4 Static and Instance Members

To this point, we have been coding our application in Main, but Main is a static member of the project class. Members of a class are either static members or instance members. Static class members are used to create data and functions that can be accessed without creating an instance of the class. Static class members can be used to separate data and behavior that is independent of any instance of the class. To think of it another way static members belong to the class, itself, whereas instance members belong to an object or instance of a class. Static is really used to group items of the same members together. This is important when dealing with the garbage collection that is going on behind the scenes. Statics will not be destroyed, where instances will.

You will get compile time errors if you attempt to mix static and instance members. Since we will be using instances of hardware devices, the best approach to writing the C# applications is to create a new class that holds the actual program. Main is just used to launch the program. The above program would be re-written as follows:

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

namespace GreenLED2
{
    public class Program
    {
        public static void Main()
        {
            Debug.Print(Resources.GetString(Resources.StringResources.String1));
```

```
                App myApp = new App();
                myApp.Run();

            }

        }

        public class App
        {

            OutputPort myGreenLED;

            public void Run()
            {
                myGreenLED = new OutputPort(Pins.GREEN_LED, true);

                while (true)
                {
                    Thread.Sleep(500);
                    myGreenLED.Write(true);

                    Thread.Sleep(500);
                    myGreenLED.Write(false);

                }

            }

        }

    }
```

The new class contains the functional part of the application. The Run method is executed from Main. The instance of myGreenLED can be used by other methods or interrupt routines because it is part of an instance, not a static. We will be using the App class programming approach for many of the exercises ahead.

5.5 Exercise 5.2: Basic Input / Output

The previous exercise showed how to setup a GPIO for output. We will build on this concept to show the code for setting up a GPIO as input.

Hardware needed:

- A jumper wire to connect HDR1-1 (PY1) to HDR3-1 (HGPIO[2])

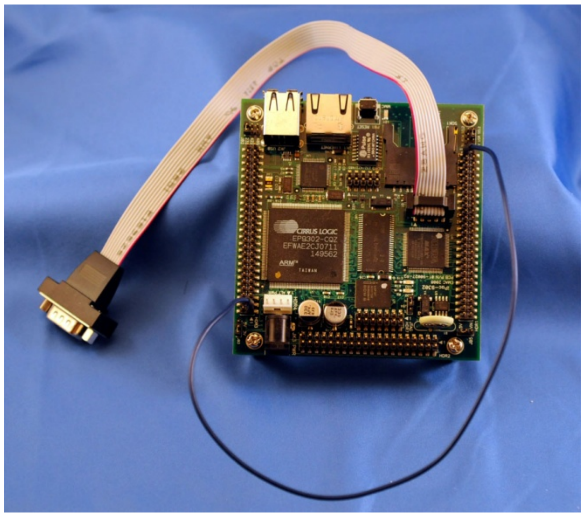
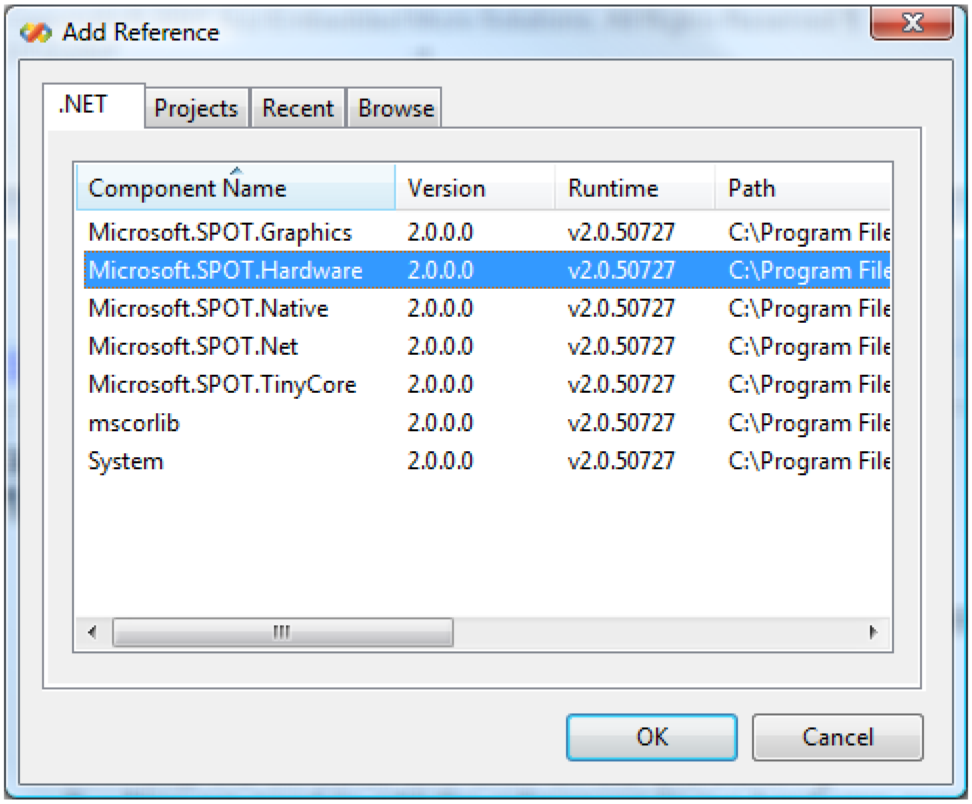


Fig 5.6 - Jumper Wire HDR1-1 to HDR3-1

Note: *If you open the project that is supplied with the EDK, due to possible directory differences, you will have to remove and re-add the SJJ_HardwareProvider resource.*

5.5.1 Create and setup the project

1. Open Visual Studio or Visual C# Express Edition.
2. From the menu, select **File**->**New**->**Project…**
3. The New Project dialog appears; expand the Project Types on the left side under **Other Languages->Visual C#**.
4. Click on **Micro Framework**
5. Click on the **Console Application** template.
6. In the Name at the bottom, type in **BasicIO**
7. Keep the defaults, and click on the **OK** button.
8. Now we need to add the references to our hardware assemblies. With the project selected in the Solution Explorer, from the menu, click on **Project**, and then select **Add Reference…** from the drop down menu items.
9. In the .NET tab of the Add Reference dialog, select **Microsoft.SPOT.Hardware**.



Component Name	Version	Runtime	Path
Microsoft.SPOT.Graphics	2.0.0.0	v2.0.50727	C:\Program File
Microsoft.SPOT.Hardware	2.0.0.0	v2.0.50727	C:\Program File
Microsoft.SPOT.Native	2.0.0.0	v2.0.50727	C:\Program File
Microsoft.SPOT.Net	2.0.0.0	v2.0.50727	C:\Program File
Microsoft.SPOT.TinyCore	2.0.0.0	v2.0.50727	C:\Program File
mscorlib	2.0.0.0	v2.0.50727	C:\Program File
System	2.0.0.0	v2.0.50727	C:\Program File

Fig 5.7 - Visual Studio Add Reference Dialog .NET Tab

10. Click **OK**.
11. With the project selected in the Solution Explorer, from the menu, click on **Project**, and then select **Add Reference…** from the drop down menu items.
12. Click on the **Browse Tab**, and locate and select the **SJJ_HardwareProvider.DLL**
13. Click **OK**.

5.5.2 Adding the code

1. From Solution Explorer, double-click on **Program.CS** to open it in the editor.
2. Add the following *using* directives to the top of the code listing:

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

3. After the Program Class. Create a new Class called App.

```
public class App
{

}
```

4. Within the App Class, let's define an output port using the PY7 pin, an input port using the HGPIO[2] pin, and an output port for the Green LED.

```
OutputPort myPY7;
InputPort myHGPIO2;
```

```
OutputPort myGreenLED;
```

***Warning**: If a GPIO port is incorrectly defined, for example, defining a port as Output when it is input-only, NO error will be reported back by CLR. The application will fail outright, and you will have to use MFDeploy to erase the application from flash before you can deploy a new, corrected version.*

5. Create a new method called Run:

```
public void Run()
{

}
```

6. In the Run method, create the new instances of the ports.

```
myPY7 = new OutputPort(Pins.GPIO_PORT_Y_7, false);
myHGPIO2 = new InputPort(Pins.HGPIO2_HDR3_1, false, Port.ResistorMode.Disabled);
myGreenLED = new OutputPort(Pins.GREEN_LED, false);
```

7. In the Run method, add a while-loop as follows:

```
while (true)
{

    Thread.Sleep(500);
    myPY7.Write(true);

    Thread.Sleep(10);

    myGreenLED.Write(myHGPIO2.Read());

    Thread.Sleep(10);

    myPY7.Write(false);

    Thread.Sleep(10);

    myGreenLED.Write(myHGPIO2.Read());

}
```

GPIO pin PY7 will output a 1 or a 0. GPIO Pin HGPIO[2] will read the input. The Boolean result from the myHGPIO2.Read will be used to set the output state for the GreenLED. The green LED will flash on and off, tracking the state of myPY7. If the jumper between PY7 and HGPIO[2] is removed the green LED stops flashing.

8. In the Main method add the following to run the application:

```
App myApp = new App();
myApp.Run();
```

9. Save the project

The application should look like the following:

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

namespace BasicIO
{
    public class Program
    {
        public static void Main()
        {
            Debug.Print(Resources.GetString(Resources.StringResources.String1));
            App myApp = new App();
            myApp.Run();

        }

    }

    public class App
    {

        OutputPort myPY7;
        InputPort myHGPIO2;
        OutputPort myGreenLED;

        public void Run()
        {
           //Connect a jumper wire between PY7 and HGPIO2
           myPY7 = new OutputPort(Pins.GPIO_PORT_Y_7, false);
           myHGPIO2 = new InputPort(Pins.HGPIO2_HDR3_1, false,
           Port.ResistorMode.Disabled);
           myGreenLED = new OutputPort(Pins.GREEN_LED, false);

           while (true)
           {

                Thread.Sleep(500);
                myPY7.Write(true);

                Thread.Sleep(10);

                myGreenLED.Write(myHGPIO2.Read());

                Thread.Sleep(10);

                myPY7.Write(false);

                Thread.Sleep(10);

                myGreenLED.Write(myHGPIO2.Read());

           }

        }

    }

}
```

5.5.3 Configure Project Properties, Build, and Deploy the application.

1. Make sure the Null modem cable is connected to the iPac-9302 and the development computer.
2. Using a jumper wire, jumper HDR1-1 (PY7) to HDR3-1 (HGPIO[2]).
3. With the project selected in the Solution Explorer, from the menu, select **Project** and then select **BasicIO Properties…** from the submenu.
4. The project properties page appears, click on the **Micro Framework** tab.

5. Under Deployment, select **Serial** for the Transport and **Com#** as the Device. Where # is the COM port number of the development computer.
6. Save the project
7. From the menu, select **Build**, and then select **Build BasicIO** from the drop down menu.
8. Make sure the iPac-9302 is powered on. You need to wait at least 20 seconds before deploying an application. You can also monitor the boot-up of the iPac-9302 with MFDeploy and use the clearing of the BootLoader Flag to shorten the boot time, but **remember to disconnect MFDeploy before you attempt to deploy an application from Visual Studio.**
9. From the menu, select **Build**, and then select **Deploy BasicIO**.
10. The output window should show a successful download. Power down the board, wait one second, and power up the board.

After about 6 seconds, the onboard green LED should start flashing on and off. Try disconnecting the jumper wire, and notice that the green LED doesn't flash.

5.6 Exercise 5.3: Polling GPIO Pins

There are situations were a program needs to wait on a GPIO pin state to perform an action. As we already mentioned, the iPac-9302 comes with GPIOs that are interrupt capable, but many of the GPIOs don't have this feature. Polling can be used to wait on a state of non-interrupt capable GPIOs.

In this exercise we are going to take advantage of a hardware characteristic in the Cirrus EP9302 processor. When an input pin of the Cirrus EP9302 SOC is left floating (not tied to anything), the input is pulled high (Note that the CPLD inputs behave differently). We will use a wire attached to a GPIO input port of the Cirrus EP9302 SOC as a probe to touch ground (GND) as the signal to perform the alternative function of the exercise.

Hardware needed:

- A jumper wire

Note: *If you open the project that is supplied with the EDK, due to possible directory differences, you will have to remove and re-add the SJJ_HardwareProvider resource.*

5.6.1 Create and setup the project

1. Open Visual Studio or Visual C# Express Edition.
2. From the menu, select **File**->**New**->**Project…**
3. The New Project dialog appears; expand the Project Types on the left side under **Other Languages->Visual C#**.
4. Click on **Micro Framework**
5. Click on the **Console Application** template.
6. In the Name at the bottom, type in **PollingGPIO**
7. Keep the defaults, and click on the **OK** button.
8. Now we need to add the references to our hardware assemblies. With the project selected in the Solution Explorer, from the menu, click on **Project**, and then select **Add Reference…** from the drop down menu items.
9. In the .NET tab of the Add Reference dialog, select **Microsoft.SPOT.Hardware**.
10. Click **OK**.
11. From the menu, click on **Project**, and then select **Add Reference…** from the drop down menu items.

12. Click on the Browse Tab, and locate and select the **SJJ_HardwareProvider.DLL**
13. Click **OK**
14. Save the project

5.6.2 Adding the code

1. From Solution Explorer, double-click on **Program.CS** to open it in the editor.
2. Add the following *using* directives to the top of the code listing:

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

3. After the Program Class. Create a new Class called App.

```
public class App
{

}
```

4. Within the App Class, let's define an input port for the HGPIO[2] pin and an output port for the Green LED.

```
InputPort myHGPIO2;
OutputPort myGreenLED;
```

5. Create a new method called Run:

```
public void Run()
{

}
```

6. In the Run method, create the new instances of the ports.

```
myHGPIO2 = new InputPort(Pins.HGPIO2_HDR3_1, false, Port.ResistorMode.Disabled);

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

7. In the Run method, add a while-loop that polls on the reading of the state of HGPIO[2]

```
while (true)
{

    if (myHGPIO2.Read() == false)
    {
        myGreenLED.Write(true);
    }
    else
    {
        myGreenLED.Write(false);
    }
}
```

8. In the Main method add the following to run the application:

```
App myApp = new App();
myApp.Run();
```

9. Save the project.

The application should look like the following:

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

namespace PollingGPIO
{
    public class Program
    {
        public static void Main()
        {
            Debug.Print(Resources.GetString(Resources.StringResources.String1));
            App myApp = new App();
            myApp.Run();


        }

    }

    public class App
    {

        InputPort myHGPIO2;
        OutputPort myGreenLED;

        public void Run()
        {
           myHGPIO2 = new InputPort(Pins.HGPIO2_HDR3_1, false,
           Port.ResistorMode.Disabled);
           myGreenLED = new OutputPort(Pins.GREEN_LED, false);

           while (true)
           {

                if (myHGPIO2.Read() == false)
                {
                    myGreenLED.Write(true);
                }
                else
                {
                    myGreenLED.Write(false);
                }

           }

        }

    }

}
```

5.6.3 Configure Project Properties, Build, and Deploy the application.

One end of the jumper is connected to the GPIO pin. The other is used as a probe to touch and GND the signal.

1. Make sure the Null modem cable is connected to the iPac-9302 and the development computer.

2. Using a jumper wire, connect one end to HDR3-1 (HGPIO[2]).
3. With the project selected in the Solution Explorer, from the menu, select **Project** and then select **PollingGPIO Properties…** from the submenu.
4. The project properties page appears, click on the **Micro Framework** tab.
5. Under Deployment, select **Serial** for the Transport and **Com#** as the Device. Where # is the COM port number of the development computer.
6. Save the project
7. From the menu, select **Build**, and then select **Build PollingGPIO** from the drop down menu.
8. Make sure the iPac-9302 is powered on. You need to wait at least 20 seconds before deploying an application. This gives TinyBooter a chance to turn control over to the CLR. You can also monitor the boot-up of the iPac-9302 with MFDeploy and use the clearing of the BootLoader Flag to shorten the boot time, but **remember to disconnect MFDeploy before you attempt to deploy an application from Visual Studio.**
9. From the menu, select **Build**, and then select **Deploy PollingGPIO**.
10. The output window should show a successful download. Power down the board, wait one second, and power up the board.
11. Wait until the Green LED goes out.
12. Using the other end of the jumper, touch any of the GND pins on HDR3 and then remove. The Green LED will go one when you touch the end of the jumper wire to a ground pin and will stay on until you remove it from the GND pin.

The while-loop is continuous, so if you would only want a single action to occur on the GPIO signal, then using interrupts are the better solution.

5.7 Exercise 5.4: GPIO Interrupt

Interrupts break the flow of execution so something else can run and respond to an event. This eliminates having to write polling code loops that must run in separate threads, and provides a more timely response to events. There are 14 GPIOs that support interrupts. The InterruptPort class in .NET MF not only has properties and methods, but it also has an event handler that is used to attach your event response code to the occurrence of the GPIO interrupt.

The advantage that GPIO interrupts have over polling is that interrupts only occur and the interrupt response code only runs on the instance of a signal change on a GPIO pin. You can immediately appreciate the economy of CPU usage to handle an event, if it is handled as an interrupt, rather than having the CPU polling for the event condition. In the polling example, the main program had to have a loop that repeatedly had to sample the GPIO input and decide if an event had occurred. Thus, whether an event occurs or not, the CPU is busy with event handling code all the time. In a real-world application where the application is doing more things than just polling for a GPIO pin state change and responding to it, the GPIO event polling would be put in its own thread, and the other application functions would be put in one or more separate threads. The CPU would be much busier switching among threads and polling the GPIO states, than it would be if it could concentrate on activities other than GPIO state handling until an actual GPIO state changed occurred. In this example, the main code does not have to run any event-specific code until an actual event occurs. The signal change is detected by the hardware and the system's interrupt service routine is spawned. The system's interrupt service routine is connected to your event handler when you create the InterruptPort object, and it receives control when the .NET MF CLR schedules a thread switch after a GPIO interrupt event has been queued.

The signal change can be rising edge or falling edge. The iPac-9302 does not support level sensitive interrupts.

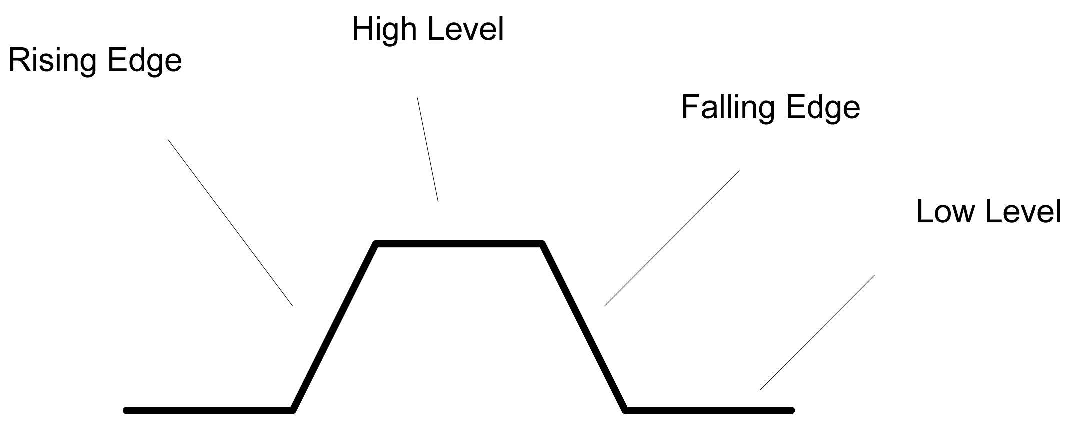


Fig 5.8 - Anatomy of an Interrupt Pulse

This exercise will be similar to the last exercise, but this time we will be using an interrupt to trigger a response to turn on the LED.

Interrupts in .NET MF applications run in an Interrupt Service Routine (ISR), which is intended to perform small atomic operations. In more complex operating systems, such as Windows CE and TenAsys® INtime®, ISRs spawn Interrupt Service Threads (IST), which handle the bulk of the processing. Since .NET MF only offers ISRs, you don't want your service routines to be long or take up too much time. If there is too much processing in the interrupt service routine, the system will fail to see other interrupts, because interrupts are blocked when the ISR is running. The .NET MF CLR does not support nesting of interrupts at this time, so interrupts could be missed. In the extreme case of a long interrupt service routine and frequently recurring interrupts, the CPU can be completely consumed with running the interrupt service routine, over and over, and the main program will be starved out and not get any execution time at all. You will want to avoid this, so keep your event handlers lean and mean.

Since interrupt pins can be used for things like keypads, the GPIOs also support glitch filtering to provide de-bouncing of keys with mechanical contacts. The .NET MF API setup for the interrupt pin provides an option for this glitch filtering. If glitch filtering is turned on, you will only be able to interrupt the pin at a maximum rate of 50Hz. This is because a time delay is inserted in the GPIO input response to provide the glitch filtering or de-bouncing. Disable the glitch filter option if you need high frequency interrupts and do not have mechanical contacts directly providing the GPIO input signal switching.

One very important item to understand is that Interrupts in .NET MF will NOT preempt the currently running thread of the system. The interrupt will be queued until the CLR scheduler next runs. Either the current task or thread must have an explicit thread.sleep or other activity that would put the currently running threads to sleep and cause the CLR scheduler to run. If all threads are in the sleep state when an interrupt occurs, the interrupt will be handled immediately. If the scheduler is run, due to a Thread.Sleep or similar activity, the interrupt will run as soon as the scheduler runs. If a thread is continuously active for more than 20 mSec, Then the interrupt will have to wait for the 20mSec system timer to expire and the CLR scheduler to run for round-robin scheduling. This is very important when it comes to architecting the application; especially when high-frequency interrupts are coming in on the GPIO line. Put simply, the .NET MF is not a real-time operating system. If you have repetitive GPIO events that occur at a frequency higher than 50 Hz, you are going to miss interrupts periodically, and the system should be designed to recover from missed interrupts.

Hardware needed:

- A jumper wire

***Note:** If you open the project that is supplied with the EDK, due to possible directory differences, you will have to remove and re-add the SJJ_HardwareProvider resource.*

5.7.1 Create and setup the project

1. Open Visual Studio or Visual C# Express Edition.
2. From the menu, select **File**->**New**->**Project…**
3. The New Project dialog appears; expand the Project Types on the left side under **Other Languages->Visual C#**.
4. Click on **Micro Framework**
5. Click on the **Console Application** template.
6. In the Name at the bottom, type in **InterruptGPIO**
7. Keep the defaults, and click on the **OK** button.
8. Now we need to add the references to our hardware assemblies. With the project selected in the Solution Explorer, from the menu, click on **Project**, and then select **Add Reference…** from the drop down menu items.
9. In the .NET tab of the Add Reference dialog, select **Microsoft.SPOT.Hardware**.
10. Click **OK**.
11. From the menu, click on **Project**, and then select **Add Reference…** from the drop down menu items.
12. Click on the Browse Tab, locate and select the **SJJ_HardwareProvider.DLL.**
13. Click **OK**
14. Save the project

5.7.2 Adding the code

1. From Solution Explorer, double-click on **Program.CS** to open it in the editor.
2. Add the following *using* directives to the top of the code listing:

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

3. After the Program Class. Create a new Class called App.

```
public class App
{

}
```

4. Within the App Class, let's define an interrupt port for the EGPIO[1] pin, an output port for the PY7 pin, and an output port for the Green LED.

```
OutputPort myGreenLED;
InterruptPort EGPIO1;
OutputPort myPY7;
```

5. Create a new method called Run:

```
public void Run()
{

}
```

6. In the Run method, create the new instances of the output ports, and then set them both low.

```
myGreenLED = new OutputPort(Pins.GREEN_LED, false);
myPY7 = new OutputPort(Pins.GPIO_PORT_Y_7, false);
```

7. Only the EGPIOs and the FGPIO can be used for interrupts. EGPIO[1] will be used for the interrupt detection on a rising edge. Create an interrupt port just like an input or output port:

```
EGPIO1 = new InterruptPort(Pins.EGPIO1_HDR3_13_CLK1_HZ, true,
Port.ResistorMode.Disabled, InterruptModes.InterruptEdgeHigh);
```

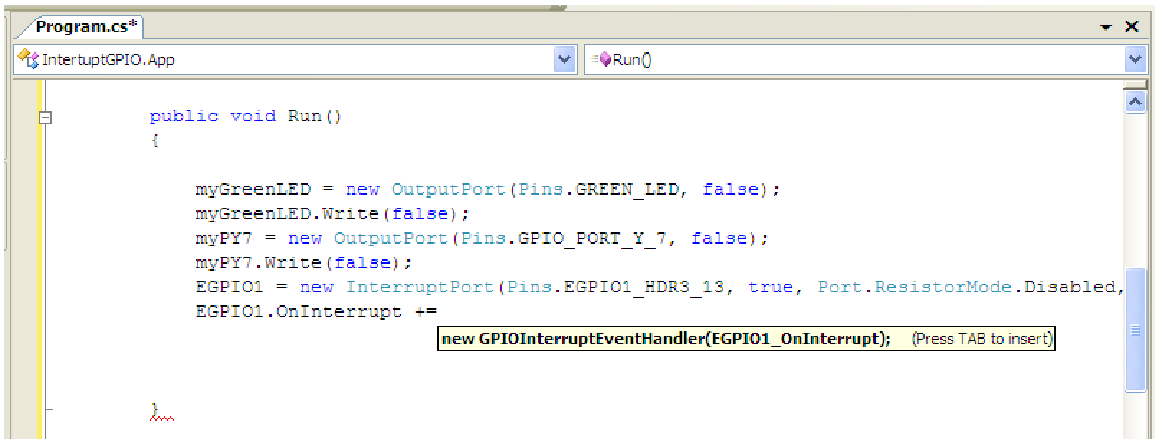
Notice that we use InterruptMode from the SJJ Hardware provider, and not Port.InterruptMode. The iPac-9302 does support positive and negative edge interrupts; it only supports one or the other, not both. Also, level sensitive interrupts are not supported.

***Note**: The "true" setting in the second argument of the constructor turns on the glitch filter to help with de-bouncing on the interrupt pin. If glitch filtering is enabled, you will only be able to interrupt the pin at a maximum rate of 50Hz. If you need high frequency interrupts and do not have mechanical contacts switching the interrupt signal, make this setting false. Turning on the glitch filter in this exercise is only for example purposes. It is not required since we are driving the interrupt GPIO input with a GPIO output, not a mechanical contact switch, so there will not be any contact bounce.*

8. Now comes the tricky part, adding the event handler. Type in the following:

```
myEGPIO1.OnInterrupt +=
```

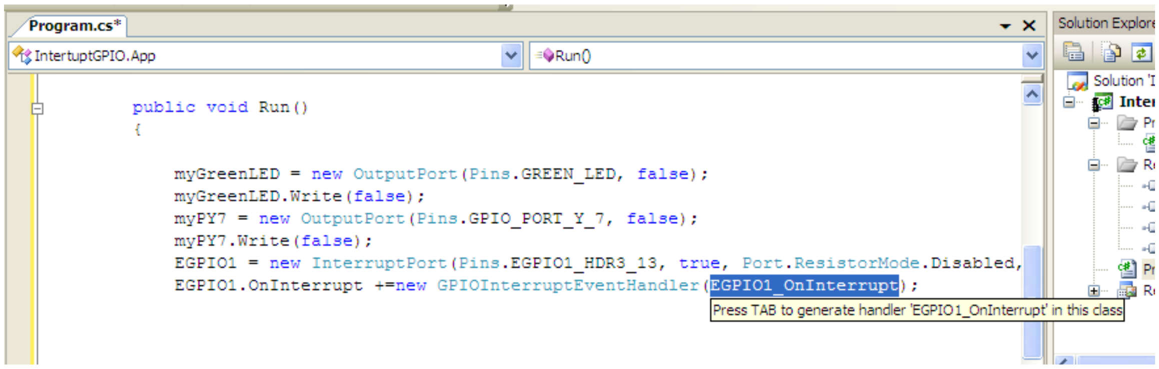
9. The Intellisense will suggest that you hit **TAB** to insert the rest of the line, do so and click **TAB**.



```
        public void Run()
        {

            myGreenLED = new OutputPort(Pins.GREEN_LED, false);
            myGreenLED.Write(false);
            myPY7 = new OutputPort(Pins.GPIO_PORT_Y_7, false);
            myPY7.Write(false);
            EGPIO1 = new InterruptPort(Pins.EGPIO1_HDR3_13, true, Port.ResistorMode.Disabled,
            EGPIO1.OnInterrupt +=
                          new GPIOInterruptEventHandler(EGPIO1_OnInterrupt);   (Press TAB to insert)


        }
```

Fig 5.9 - Using Intellisense to Create an Event Handler

10. Then click **TAB** again to create the Interrupt handler itself.



```
public void Run()
{

    myGreenLED = new OutputPort(Pins.GREEN_LED, false);
    myGreenLED.Write(false);
    myPY7 = new OutputPort(Pins.GPIO_PORT_Y_7, false);
    myPY7.Write(false);
    EGPIO1 = new InterruptPort(Pins.EGPIO1_HDR3_13, true, Port.ResistorMode.Disabled,
    EGPIO1.OnInterrupt +=new GPIOInterruptEventHandler(EGPIO1_OnInterrupt);
                                                      Press TAB to generate handler 'EGPIO1_OnInterrupt' in this class
```

Fig 5.10 - Completing the Event Handler Creation

11. In the run method, add the following after the EGPIO.OnInterrupt line:

```
while (true)
{

    Thread.Sleep(500);
    myGreenLED.Write(false);
    Thread.Sleep(500);
    myPY7.Write(true);//Trigger an interrupt

}
```

12. In the EGPIO1_OnInterrupt method, add the following:

```
myGreenLED.Write(true);
myPY7.Write(false); // Go low for the next interrupt
```

As you can see we keep things simple and short in the Interrupt routine.

13. In the Main method add the following to run the application:

```
App myApp = new App();
myApp.Run();
```

14. Save the project.

The code should look like the following:

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

namespace IntertuptGPIO
{
    public class Program
    {
        public static void Main()
        {
            Debug.Print(Resources.GetString(Resources.StringResources.String1));
            App myApp = new App();
            myApp.Run();

        }
    }

    public class App
    {

        OutputPort myGreenLED;
        InterruptPort EGPIO1;
        OutputPort myPY7;

        public void Run()
        {

            myGreenLED = new OutputPort(Pins.GREEN_LED, false);
            myPY7 = new OutputPort(Pins.GPIO_PORT_Y_7, false);
            EGPIO1 = new InterruptPort(Pins.EGPIO1_HDR3_13, true,
            Port.ResistorMode.Disabled, InterruptMode.InterruptPositiveEdge);
            EGPIO1.OnInterrupt += new
           GPIOInterruptEventHandler(EGPIO1_OnInterrupt);

           while (true)
           {

                Thread.Sleep(500);
                myGreenLED.Write(false);
                Thread.Sleep(500);
                myPY7.Write(true);//Trigger an interrupt

           }

        }

        void EGPIO1_OnInterrupt(Cpu.Pin port, bool state, TimeSpan time)
        {
            myGreenLED.Write(true);
            myPY7.Write(false); // Go low for the next interrupt

        }

    }


}
```

5.7.3 Configure Project Properties, Build, and Deploy the application.

One end of the jumper is connected to the GPIO pin HDR3-13 (EGPIO[1]), and the other is connected to the GPIO pin HDR1-1 (PY7).

1. Make sure the Null modem cable is connected to the iPac-9302 and the development computer.
2. Using a jumper wire, connect one end to HDR3-13 (EGPIO[1]), and the other end to HDR1-1 (PY7).
3. With the project selected in the Solution Explorer, from the menu, select **Project** and then select **InterruptGPIO Properties…** from the submenu.
4. The project properties page appears, click on the **Micro Framework** tab.
5. Under Deployment, select **Serial** for the Transport and **Com#** as the Device. Where # is the COM port number of the development computer.
6. Save the project
7. From the menu, select **Build**, and then select **Build InterruptGPIO** from the drop down menu.
8. Make sure the iPac-9302 is powered on. You need to wait at least 20 seconds before deploying an application. This gives TinyBooter a chance to turn control over to the CLR. You can also monitor the boot-up of the iPac-9302 with MFDeploy and use the clearing of the BootLoader Flag to shorten the boot time, but **remember to disconnect MFDeploy before you attempt to deploy an application from Visual Studio.**

9. From the menu, select **Build**, and then select **Deploy InterruptGPIO**.
10. The output window should show a successful download. Power down the board, wait one second, and power up the board. The Green LED should start flashing periodically since it is getting an interrupt every cycle through the while-loop.

5.7.4 Debugging the Interrupt

In order to debug an application with interrupts, you have to place a break point in the interrupt routine itself. If you were to place a break point in the main program while-loop and then hit the break point, you would never jump into the OnInterrupt code.

1. Set a break point in the while-loop.
2. Make sure that the Solution Configuration is set to Debug.
3. Power cycle the board
4. Hit F5; or from the Debug menu, click **Start Debugging**.
5. If the application has not already been deployed, Visual Studio will deploy it and then launch the debugger and run the application in the debug mode. You should hit the break point. Step over the instructions until you hit the myPY7.Write(true), and hit step over again. The Interrupt will be called, but the debugger will never jump to the code.
6. Stop debugging.
7. Now set a second break point in the Interrupt routine.
8. Power cycle the board
9. Start Debugging.
10. Step through the code. This time when you step over the myPY7.Write(true) instruction, you should hit the second breakpoint in the Interrupt routine, and you can debug through the code.
11. Stop debugging.
12. Close and save the project.

Note: *when you are debugging through code with interrupt handlers, the state of the device after handling a breakpoint will not be the same as that state would have been with a free-running system. You have to be judicious about where you place your breakpoints and how far you can step through the code after hitting a breakpoint before stopping the debug session and repositioning the breakpoints.*

5.8 Exercise 5.5: Interrupts and Architecture

Now let's see how interrupt timing can affect an application. A frequency generator will be needed to generate TTL input to FGPIO1. An O-scope will be needed to see the effects of high frequency interrupts.

Note: *If you open the project that is supplied with the EDK, due to possible directory differences, you will have to remove and re-add the SJJ_HardwareProvider resource.*

1. Open the BasicProcessing application from the last chapter
2. Uncomment the Debug.Print lines in each of the 4 methods and the Main for-loops.
3. Build and download the application again.
4. Using SJJ_COMM Lite, connect to the COM port.
5. Power cycle the iPac-9302.
6. Connect a frequency generator to generate TTL input to FGPIO1, start the generator and set it for about 30Hz. DO NOT INTERRUPT AT A HIGHER RATE. Because of the Debug.Print the application will only handle interrupts at or below this frequency, or the application will never processes anything else.

7. Power cycle the iPac-9302. You will see the Debug.Print for the interrupt in the output. Notice that the debug prints only occur between our other threads. The Interrupt does not preempt the current process in action, but is scheduled in when the CLR switches from one thread to the next.

```
:
:
Thread 1 27
Thread 1 28
Thread 1 29
[FGPIO1] Hardware Interrupt
Thread 3 30
Thread 3 31
Thread 3 32
Thread 3 33
Thread 3 34
Thread 3 35
Thread 3 36
Thread 3 37
Thread 3 38
Thread 3 39
[FGPIO1] Hardware Interrupt
Thread 4 20
Thread 4 21
.
.
.
```

Debug.Print affects the timing of the system so you may have to use other means to debug timing issues. In this application Debug.Print affects the timing of each thread, so some threads may loop in different order, but the threads will finish based on their priority.

```
.
.
.
Thread 4 46
Thread 4 47
Thread 4 48
Thread 4 49
[FGPIO1] Hardware Interrupt
[Thread 3] completed
[Thread 2] completed
[Thread 1] completed
[Thread 4] completed
Done.
```

If you are curious about what happens with a higher interrupt frequency, comment out the Debug.Print line in FGPIO1_OnInterrupt, and uncomment the lines that use EGPIO10 as an output pin. Attach an O-scope to watch the output from EGPIO10. The higher you go with the frequency the closer you are to having CLR only service interrupts. A key point here is that you want your interrupt callbacks to be as short as possible. Once again, it is very important to architect the system carefully so you don't starve out the rest of your application.

5.9 Exercise 5.6: The 555 Timer Project

The 555 Timer also known as "The IC Time Machine" has been around since 1971. It is a versatile and stabile timer IC that supports three operating modes:

- Monostable – single pulse trigger operations.
- Astable – Free running mode that acts as an oscillator.
- Bistable – Act as a Flip-Flop.

The past projects have been using what is available on the iPac-9302. Now, we will start connecting external circuits to the board, and in this case, we will use the 555 Timer as periodic interrupt source.

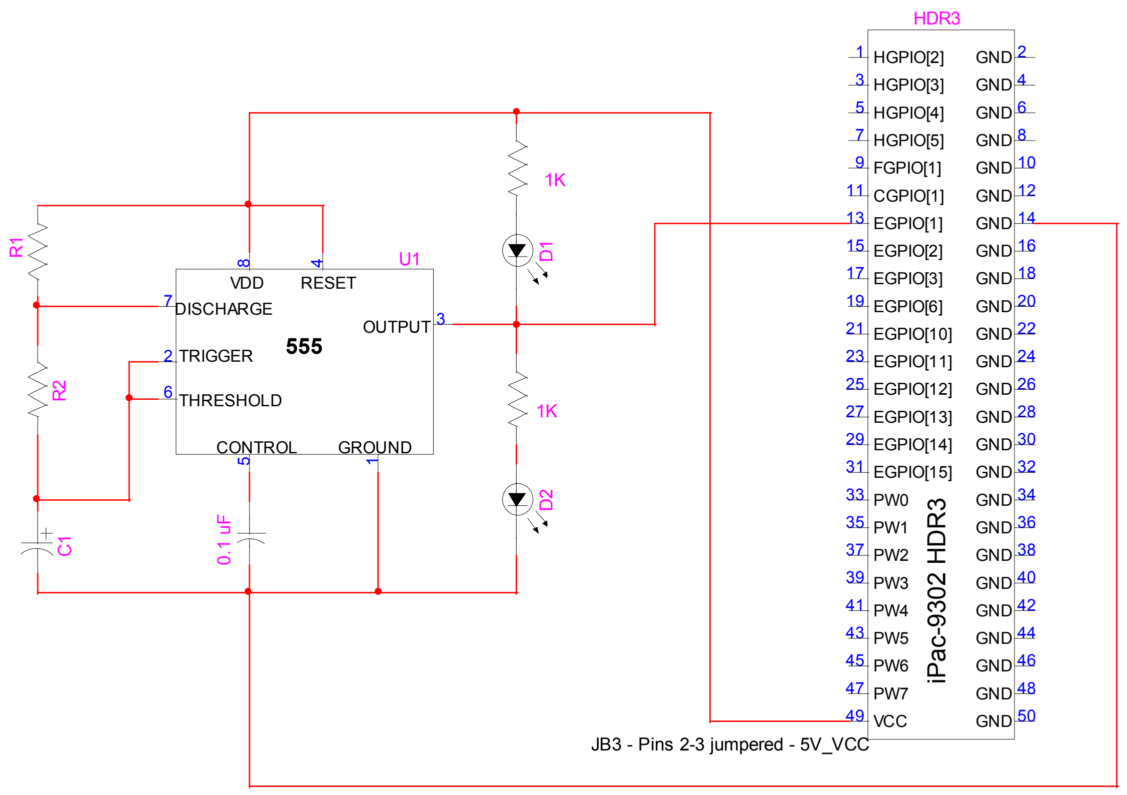


Fig 5.11 - 555 Timer as Interrupt Source

The following equations can be used to help choose the R1, R2, and C1 for the astable mode:

- Positive Time Interval (T1) = 0.693 * (R1+R2) * C1
- Negative Time Interval (T2) = 0.693 * R2 * C1
- Frequency = 1.44 / ((R1+2R2) * C1

Hardware required:

R1 = 1K
R2 = 10K
C1 = 10uF
0.1uF
555 Timer IC
Bread board
Wire
5V DC Power Supply

Optional:

D1 = D2 = 2 LEDs
2 1K Resistors

With the recommended R1, R2, and C1 above are used, the frequency should be about 6 Hz depending on the component tolerances. You may want to add LEDs to the output of the 555 circuit to allow you to visually see the output line toggling.

Note: *If you open the project that is supplied with the EDK, due to possible directory differences, you will have to remove and re-add the SJJ_HardwareProvider resource.*

5.9.1 Creating and Test the 555 Circuit

1. Make sure that you have completed Exercise 5.4: GPIO Interrupt.
2. Disconnect power from the iPac-9302.
3. Use bread a board to create the 555 timer circuit, but do not connect it to the iPac-9302. It is good practice to test a circuit like this before connecting the iPac-9302 to prevent damaging the EP9302 SOC inadvertently.
4. Use a separate 5V DC power supply and an O-scope or frequency counter to test the output from the 555 timer. If you have the LEDs connected, you should see them toggle. If the LEDs are on but dim, you will need to lower the 555 timer's output frequency.

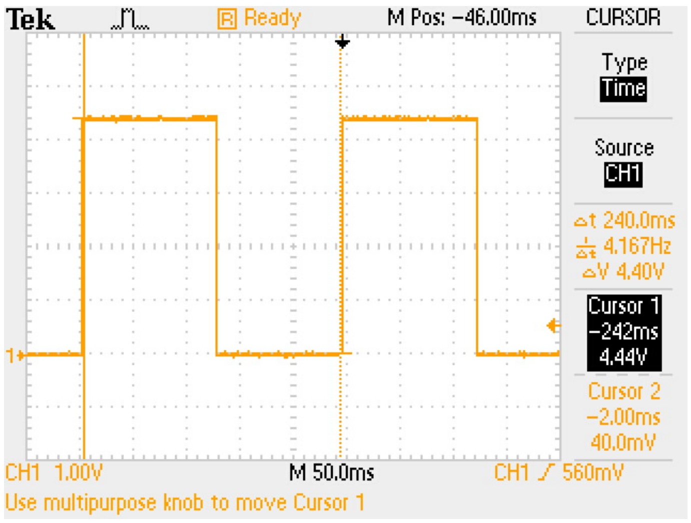


Fig 5.12 - 555 Timer Output Waveform

5. If the 555 timer is working as expected, disconnect the power supply.

Note*: The output from the 555 Timer circuit is going to be +5V to 0V. Even though the Cirrus EP9302 is a 3.3V processor, the pins are 5v tolerant. You can connect TTL devices directly to the processor's GPIOs.*

6. Connect the 555 Timer circuit's GND to the iPac-9302 HDR3-14.
7. Connect the 555 Timer circuit's Power to the iPac-9302 's HDR3-49, which should be set to 5 volts by the iPac-9302 jumper JB3.
8. Remove the jumper wire if it is still connected to PY7 and EGPIO[1].
9. Connect the output of the 555 timer circuit to the HDR3-13 (EGPIO[1]) pin of the iPac-9302.

5.9.2 Modify the Exercise 5.4: GPIO Interrupt Application and Run the Application

Now we need to modify the previous Exercise.

1. Open the InterruptGPIO project.
2. Comment out the all references to myPY7, which should be 5 lines to comment out.
3. Comment out the second Thread.sleep(500).
4. Save the project

The code should look like the following:

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

namespace IntertuptGPIO
{
    public class Program
    {
        public static void Main()
        {
            Debug.Print(Resources.GetString(Resources.StringResources.String1));
            App myApp = new App();
            myApp.Run();

        }

    }

    public class App
    {

        OutputPort myGreenLED;
        InterruptPort EGPIO1;
        //OutputPort myPY7;

        public void Run()
        {

            myGreenLED = new OutputPort(Pins.GREEN_LED, false);
            myGreenLED.Write(false);
            //myPY7 = new OutputPort(Pins.GPIO_PORT_Y_7, false);
            //myPY7.Write(false);

            EGPIO1 = new InterruptPort(Pins.EGPIO1_HDR3_13, true,
            Port.ResistorMode.Disabled, InterruptMode.InterruptPositiveEdge);
            EGPIO1.OnInterrupt += new
            GPIOInterruptEventHandler(EGPIO1_OnInterrupt);

             while (true)
             {

                Thread.Sleep(500);
                myGreenLED.Write(false);
                //Thread.Sleep(500);
                //myPY7.Write(true);//Trigger an interrupt

             }

        }

        void EGPIO1_OnInterrupt(Cpu.Pin port, bool state, TimeSpan time)
        {
            myGreenLED.Write(true);
```

```
                    //myPY7.Write(false); // Go low for the next interrupt
                }
            }

        }
```

5. Make sure the Null modem cable is connected to the iPac-9302 and the development computer.
6. From the menu, select **Build**, and then select **Build InterruptGPIO** from the drop down menu.
7. Power on the iPac-9302. You need to wait at least 20 seconds before deploying an application. This gives TinyBooter a chance to turn control over to the CLR. You can also monitor the boot-up of the iPac-9302 with MFDeploy and use the clearing of the BootLoader Flag to shorten the boot time, but **remember to disconnect MFDeploy before you attempt to deploy an application from Visual Studio.**
8. From the menu, select **Build**, and then select **Deploy InterruptGPIO**.
9. The output window should show a successful download. Power down the board, wait one second, and power up the board. The Green LED should start flashing since it is getting repetitive interrupts from the 555 Timer circuit.

***Note:** the Green LED doesn't flash at a regular rate. The CLR does not have deterministic capability, so the interrupt response will vary from interrupt to interrupt depending on the state of the CLR at the time of the interrupt. At any given instant, the CLR may be busy performing the routine maintenance of the system – memory management, garbage collection, etc.*

5.10 Summary: GPIO Overload

GPIOs are very common hardware in control systems, and the iPac-9302 has plenty of them. In fact, you can add more GPIOs via the SPI port interface using a SPI to GPIO controller chip. We talk about the SPI interface in the next chapter. The .NET MF GPIO classes offer the ability to read, write, and catch interrupt events. The best practice for writing these applications is to create an application class and use the Main static class to launch the application via the Run method.

We looked at different techniques to performing actions based on the state of an input GPIO pin – Polling and Interrupts. Interrupts were generated from another GPIO port and an external circuit. We learned to keep interrupt handler lean and mean, to maximize system performance. We will use the knowledge of GPIOs for use with SPI ports in the next chapter.

6 Serial Peripheral Interface (SPI)

The last chapter covered GPIOs. GPIOs can be used for a variety of input/output solutions and provide the basics for understanding Boolean, 0 and 1, logic. Now, we will look at another IO feature called SPI, which has some interesting capabilities. This chapter will cover the following:

- Basics of the SPI driver port.
- SPI classes.
- Interaction with a SPI LCD device.
- Developing a managed code driver.

6.1 SPI Basics – Not for Bit-Bang Anymore

The Serial Peripheral Interface Bus (SPI) bus is a synchronous serial data link standard that operates in full duplex mode. It is sometimes known as a “four wire” bus because it has four basic signals – chip select, clock, transmit, and receive. It was originally developed by Motorola.

Unlike RS232 communications, SPI allows more than one device to exist on the bus in a host/client configuration. The iPac-9302’s SPI port acts like the master-host and there can be several client devices on the bus. GPIOs combined with the SPI port’s chip select, can be used to create unique slave chip select signals for each slave device that is connected to the SPI bus. This provides the ability to connect several SPI slave devices to the iPac-9302’s SPI port and switch among them as desired.

The naming and phasing of the SPI signals vary from vendor to vendor. The iPac-9302 uses the following signals:

- SCLK1 – SPI Serial (Bit) Clock
- SSPTX1 – SPI Serial Output
- SSPRX1 – SPI Serial Input
- SFRM1 – SPI Frame Clock or Chip Select

Only one client device can talk to the master-host at a time. All SPI devices are tri-stated when not selected. Only one client device can be selected at a time, so there are no conflicts on the bus.

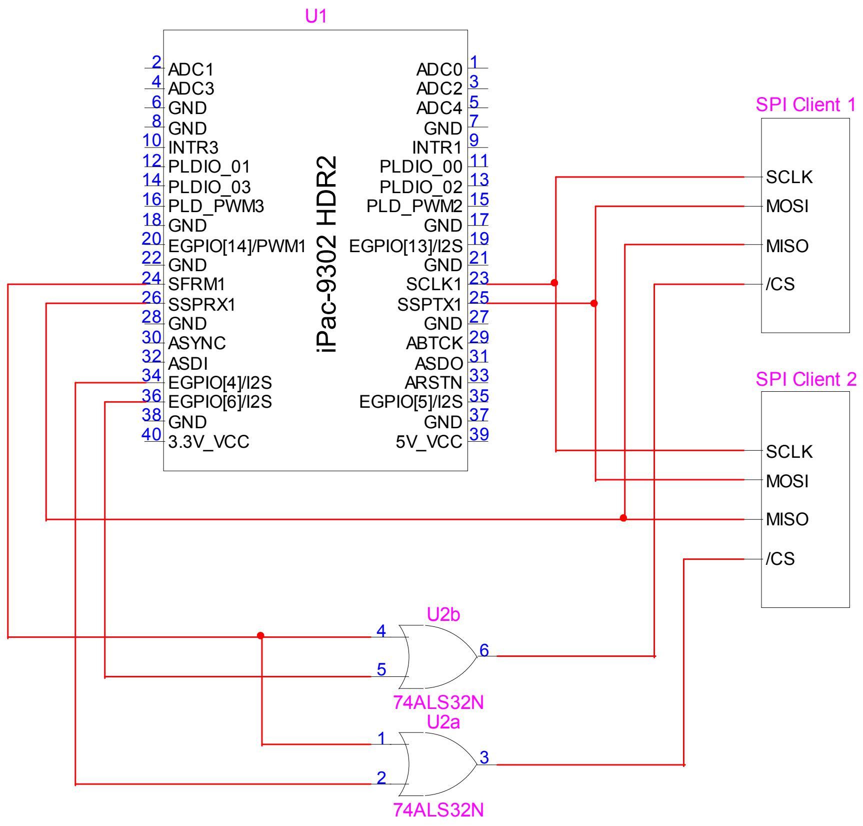


Fig 6.1 - Connecting Two SPI Client Devices

Communication starts with the master-host configuring a clock frequency. The frequency must be less than the maximum frequency that the client device can support, and the frame-to-frame time must be greater than the minimum frame-to-frame time supported by the client device. The master-host then signals a chip select low to enable the desired client device.

When a client device has been selected by the master-host, on each SPI clock cycle, a full duplex data transmission occurs. The master-host sends data on the SSPTX1 (SPI Serial Output) line, and simultaneously with each SPI clock cycle, data is sent from the client to the master-host on the SSPRX1 (SPI Serial Input) line. The data sent out on the SSPTX1 line is sent out through a shift register in the SPI controller. The client then reads the data sent on the SSPTX1 line. The client can send data back on the SSPRX1 line, but only through a write/read operation initiated and controlled by the master-host. The master-host then reads the data on the SSPRX1 (SPI Serial Input) by shifting the SSPRX1 data in and doing a serial to parallel conversion.

Clearly, it can be seen that even though the write/read operation is full duplex, any data from the client that is a response to data sent from the master-host, will be out of phase by at least one frame. It could be more depending on the processing speed of the client. There is no higher level framing protocol for SPI communications, so one has to know the particular communication characteristics used by each of the client SPI devices. If the data sent by the master-host are to be interpreted as commands to the client to which the client must respond with returned data, then it is up to the master-host to perform the required number of write/read operations following the transfer of a command, in order to get the required data back from the client. This usually requires that the master-host perform one or more dummy write/read operations, i.e. the data being sent by the master-host is ignored by the client and could be any value, but the relevant data is the data returned by the client in the write/read operation. If the SPI device is a slow device, relative to the master-host, the dummy read/writes can also simply be effectively no-operations, in which the data in both directions is ignored and the operation is used simply to generate a time delay so that the client can generate a response to the earlier command. Thus, it is not unusual to see a write/read operation that is a command being sent from the master-host to the client; the client's returned data during this operation may be meaningless and ignored. This would then be followed by a time delay dummy write/read operation, again, controlled by the master-host; the data in both directions are ignored. Finally, another dummy write/read operation would be generated by the master-host; the data to the client would be ignored but the data received back from the client would be read and processed by the master-host.

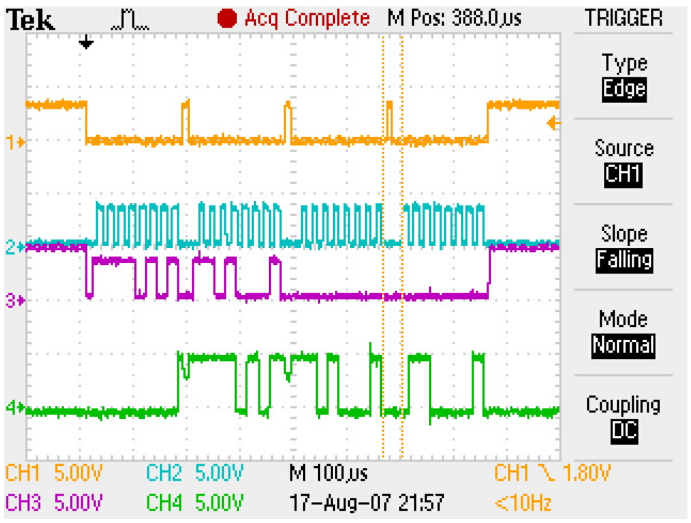

Tek
Acq Complete M Pos: 388.0µs
TRIGGER
Type
Edge
Source
CH1
Slope
Falling
Mode
Normal
Coupling
DC
1
2
3
4
CH1 5.00V CH2 5.00V M 100µs CH1 ╲ 1.80V
CH3 5.00V CH4 5.00V 17-Aug-07 21:57 <10Hz

Fig 6.2 - Typical SPI Communications Waveforms

The size of the data exchange supported by the SPI hardware can be programmed for 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 and 16-bit data transactions, but .NET MF only supports 8 or 16-bit data transactions.

Fig 6.2 above shows a basic 4-byte data exchange between the iPac-9302 master-host and the LCD SPI client device we will use in the upcoming exercises. 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, the first two bytes of which are data to the LCD SPI client. The green signal (4) is the SSPRX1 (SPI Serial Input) data, the last two bytes of which are data sent back from the LCD SPI client. Note that the data sent by the master-host in the 3rd and 4th byte exchanges are all zeroes. These are the dummy write/read operations, described earlier. The data is ignored by the LCD client, and these byte exchanges are strictly for the purpose of getting data back from the LCD client. Similarly, the data sent back by the LCD client in the first 2 byte exchanges, have no meaning to the master-host and are ignored.

Any other devices on the SPI bus that are not selected should and must ignore any transmissions on the lines and must not transmit on the SSPRX1 (SPI Serial Input) line.

Other than the physical layer, which we have discussed, there is no protocol associated with SPI communications, thus there is no overhead that would be required to support some kind of communication stack. Data can be transmitted at very high rates; clock rates up to 2 MHz are supported by the iPac-9302 SPI controller. The flip side is that management of commands and data between the managed code application on the iPac-9302 and each SPI client device must be managed on an individual device-by-device basis by the managed code application. Each device will have a different command and data exchange protocol that the managed code application must understand and adhere to. Each device may even have a different phasing between the chip select, the clock, and the data signals, which must be set up properly when the SPI object is created in the managed code application. Encapsulating the device specific protocol in a managed code driver is a method for dealing with this and will be covered later in this section.

There are many SPI devices available such as Analog-to-Digital converters, Real-Time controllers (RTC), GPIO expanders, 7-segment LED drivers, various PIC controllers, etc. In this chapter we demonstrate SPI application and managed code using a SPI-to-text LCD device.

6.2 SPI Classes

There are two SPI classes and an enumeration. The SPI.ClockRate Enumeration specifies the available SPI clock rates as a divisor of the system clock. These enumeration constants can be used for setting the clock rate in the SPI.Configuration class.

The SPI.Configuration class is used to create a new instance of the SPI interface/client. Like setting the baud rate, data and stop bits for an RS-232 port, the SPI.Configuration is use to set the following:

- ChipSelect Port – The GPIO port that goes active low when a client is going to be active.
- ChipSelect Active state – True if active high or False if it is active low
- ChipSelect Setup Time – delay between when the device is selected and the clock begins.
- ChipSelect HoldTime – defines how long the chip select stays active after the last data bit clock has been clocked onto the bus.
- Clock Idle State – True if the SPI clock is set to high while device is idle. False if the SPI click goes low while device is idle.
- Clock Edge – True – data is sampled on the SPI clock rising edge. False – data is sampled on the SPI clock falling edge.
- Clock Rate – SPI clock rate.
- SpiModule – SPI bus to be used.

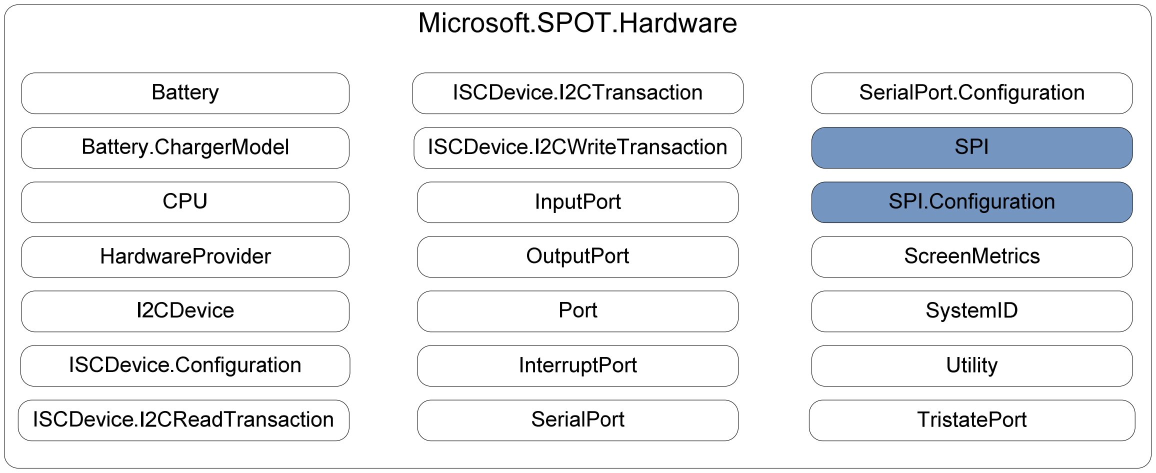


Fig 6.3 - SPI Classes

Once the instance of the configuration is setup, the SPI class is used to create an instance of the port and perform the basic SPI WriteReads and Writes. The API only supports 8 and 16 bit data transfers.

6.3 Exercise 6.1: Basic SPI LCD project

The iPac-9302 was designed for headless operations, but a serial LCD text display can be easily added using the versatile SPI port. In this exercise, we will connect a SPI LCD and write an application that sends data to display on the serial LCD.

Hardware Requirements:

- Connection Wires – 6, .025" square socket to .025 square socket – www.e-z-Hook.com
- LCD2S-162 with 2x 16 (2 line x16 Character) LCD or equivalent - http://microcontrollershop.com/

The actual SPI control board:
LCD2S-162 - Serial LCD Display Board with SPI/I2C Serial Bus for 2x16 LCDs

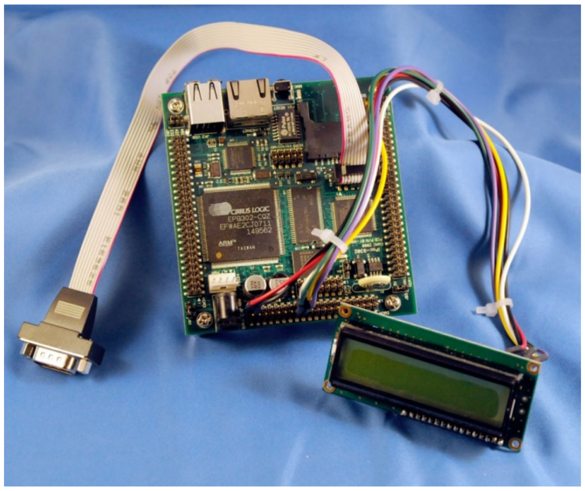


Fig 6.4 - iPac-9302 with SPI-LCD

Different combinations of the SPI controller board and Serial LCDs are available:

LCD2S 162YHA	Serial LCD Display, 2x16 characters, amber LED backlight
LCD2S 162GGN	Serial LCD Display, 2x16 characters, STN grey/blue, no backlight
LCD2S 162YGN	Serial LCD Display, 2x16 characters, STN yellow/green display
LCD2S 162BIW	Serial LCD Display, 2x16 characters, white LED backlight
LCD2S 162BIY	Serial LCD Display, 2x16 characters, yellow/green LED backlight
LCD2S 162YHY	Serial LCD Display, 2x16 pixel, yellow/green LED backlight
LCD2S 162GHB	Serial LCD, 2x16 characters, STN grey, blue LED backlight

Table 6.1 - SPI Controller and LCD Display Combinations

6.3.1 SPI Controller Commands

Gathering the SPI commands and electrical specifications are the first step to writing the application. The LCD2S-162's SPI clock frequency diagram found in section 14.4 of the Modtronix Engineering *LCD2S Revision 1, Firmware V1.10, Serial LCD Display* documentation provides the information to calculate the clock-rate needed for this application. The minimum clock pulse is 500ns high and 500ns low which means the maximum frequency is about 1 MHz, but you have to take into account the minimum time between bytes as well, which is specified to be 19 µSeconds. The data bits in a single byte require 8 clock periods. For the iPac-9302 SPI controller, the delay from the negative-going edge of the chip select to the first data bit is one clock period, and the delay from the end of the last data bit period to the negative-going edge of the next byte's chip select is .5 clock periods. Thus the byte-to-byte period for the iPac-9302 SPI controller is 9.5 clock periods. If we divide the 19 µSeconds by 9.5, we have a minimum clock period of 2 µSeconds or a maximum clock frequency of 500 KHz. Notice that the 500 KHz maximum clock rate is considerably slower than the 1 MHz clock rate that we might have naively chosen if we had only taken into consideration the minimum data bit clock timing. Even though the calculated maximum clock frequency is 500 KHz, you don't want to push the controller right to the edge; because of thermal variations and limitations on the resolution of the actually SPI clock divider circuitry, a small variance in the clock frequency could put you in the no-fly zone, so 300 KHz is a safe clock frequency maximum for this exercise.

The 2x16 LCD itself can be characterized as one long line of 32 characters or 2 lines of 16 characters each (1-16). Messages can wrap to the next line if the string sent is longer than 16 characters.

The address of each character:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Table 6.2 - LCD Display Character Addressing

The column numbers for each line:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Table 6.3 - LCD Display Column Addressing

Looking in the LCD2S-162 datasheet for the different commands, there are several byte commands to control the LCD display. All of the commands must be preceded by a 0xF5, which provides a SYNC character to the controller to let it know a command is coming.

If we simply want to write text to the first line of the LCD display, we first need to position the cursor to line 1 and the first column of that line. The command to set cursor position is 0x8A followed by the Row number (1 or 2) and a Column number (1-16). If we want to set the cursor to line 1 character 0, we would send the following:

0xF5, 0x8A, 0x01, 0x01

With the cursor positioned, we can write a string of data to the LCD. The 0x80 command followed by the string will write the data to the LCD.

0xF5, 0x80, “String”

Since SPI.Write and SPI.WriteRead are expecting Byte[] or Unint16[], the string itself will need to be reconfigured for output. Since the commands for the LCD controller are a byte in length, we will use a UTF8 Encoding to convert a string to byte characters.

The LCD2S-162 supports other LCD commands, but we will save these for the next exercise.

***Note**: It is assumed that the LCD will turn on when power is applied and display the internal Welcome message:*

Modtronix Serial
LCD Display

thus no commands are needed to enable the SPI LCD. You can check this out ahead of time by applying 5V power to the LCD2S-162.

6.3.2 Connecting the SPI LCD to the iPac-9302

The next step is to wire up the SPI LCD to the iPac-9302. The .025" square socket to .025 square socket jumper wires from www.e-z-Hook.com make for ideal connection wires. We will not use an OR-gate for the chip select, since we have only the single SPI LCD on the SPI bus; so just wire the SFRM to the CS of the SPI controller. Here are the wire signals to connect:

iPac-9302 Pin	LCD Controller Pin
SCLK1 (HDR2-23)	CLK (Connector X3-10)
SSPTX1(HDR2-25)	SDI (Connector X3-8)
SSPRX1(HDR2-26)	SD0 (Connector X3-7)
SFRM1(HDR2-24)	CS (Connector X3-9)
5V_VCC (HDR2-39)	5V (Connector X3-1, 3, or 5)
Gnd (HDR2-37)	Gnd (Connector X-6)

Table 6.4 - SPI LCD Display to iPac-9302 Connections

Warning*: If the wires are not correctly connected you could damage the device or the iPac-9302.* ***Double check all connections before you apply power****. Like the master carpenter who measures twice and cuts once, check the wiring twice and power-on once. That way the iPac-9302 will live to power-on another day.*

Here is the schematic:

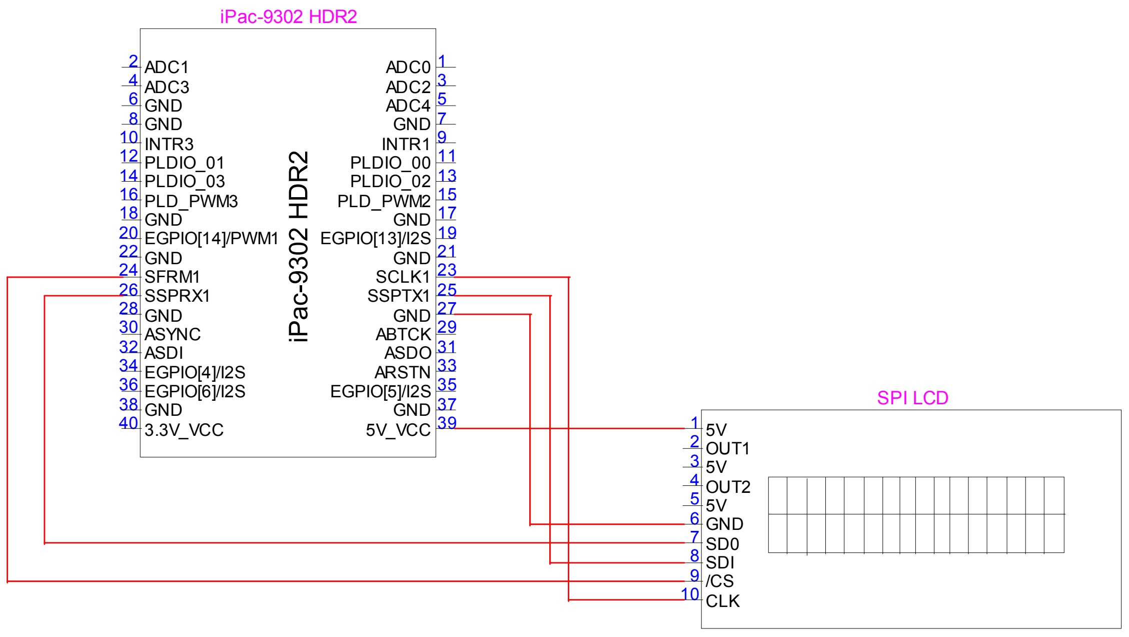


Fig 6.5 - SPI LCD Display Connection Schematic

If you power up the iPac-9302, the LCD will display its sign-on message.

6.3.3 Creating a New Application Using the SJJ_MF Console Application Template

The circuit is all set, so let's create the application. You are probably getting tired of repeating the same steps for every new application. Flexibility is one of Visual Studio's great features, and there are different ways to add on to Visual Studio. Visual Studio comes standard with the ability to export and add on project templates. There is a project template called SJJ_MF_Console Application.zip that provides the basic foundation for a .NET MF console application. We will use this template to start our application development.

Note: *If you open the project that is supplied with the EDK, due to possible directory differences, you will have to remove and re-add the SJJ_HardwareProvider resource.*

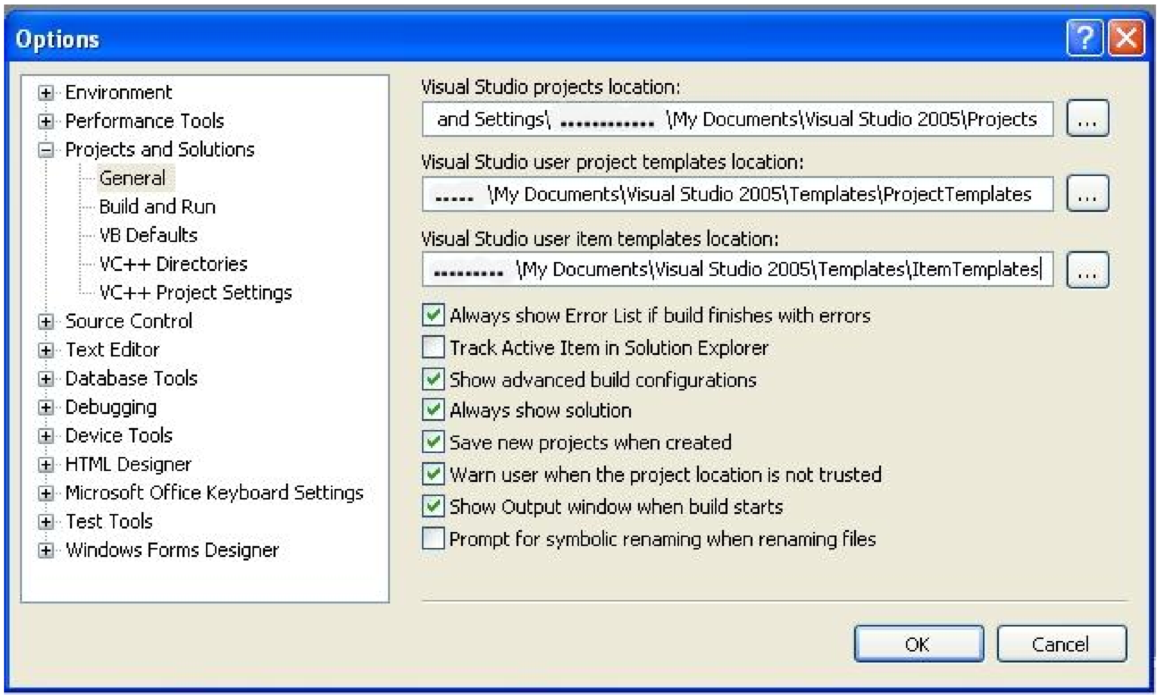
1. Locate the SJJ_MF Console Application.zip on the CD.
2. Copy the ZIP file to your Visual Studio templates folder. The folders location will depend on your operating system:

   - Windows XP Pro: \Documents and Settings\<your account>\My Documents\Visual Studio 2008\Templates\ProjectTemplates

   - Windows Vista or 7: \Users\<your account>\Documents\Visual Studio 2008\Templates\ProjectTemplates

   - For Visual C# Express: …\My Documents\Visual Studio 2008\Templates\ProjectTemplates\Visual C#

   Note: *If the template does not show up, then check the path settings for your installation by selecting* ***Tools->Options…*** *and then look at the "Visual Studio user project templates location:"*



Visual Studio projects location:
and Settings\ ………… \My Documents\Visual Studio 2005\Projects

Visual Studio user project templates location:
….. \My Documents\Visual Studio 2005\Templates\ProjectTemplates

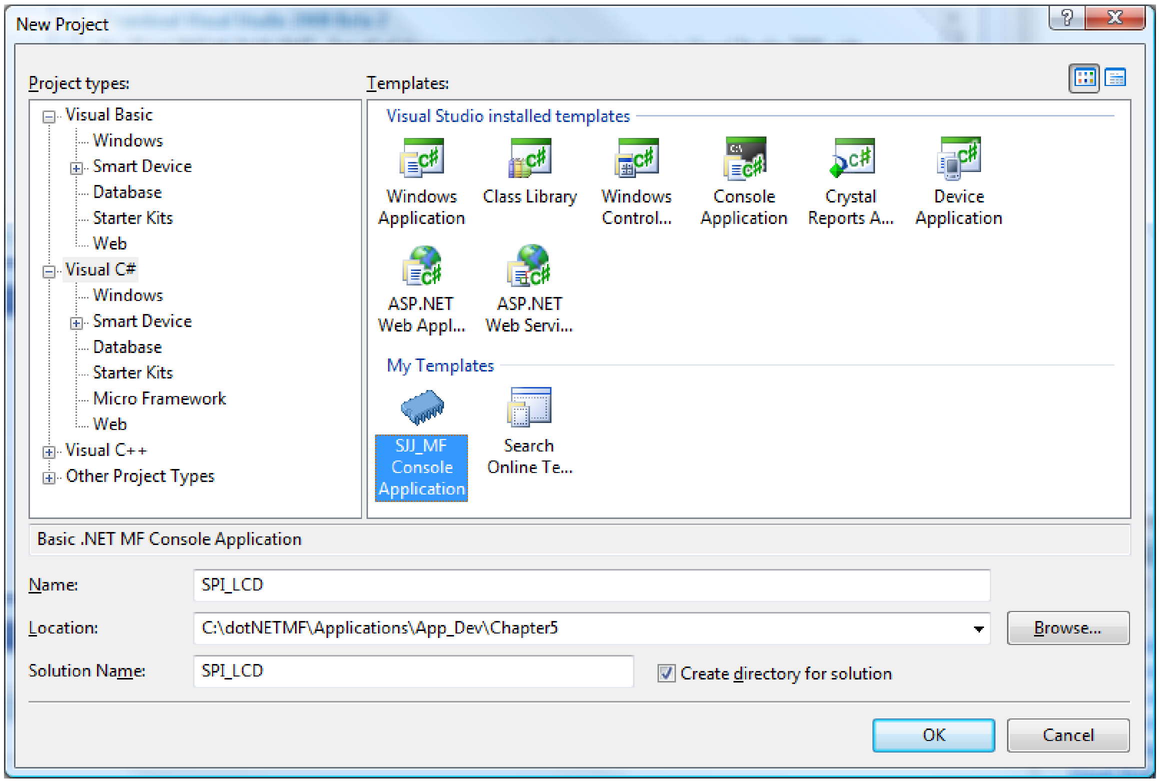
Visual Studio user item templates location:
……… \My Documents\Visual Studio 2005\Templates\ItemTemplates

- [✓] Always show Error List if build finishes with errors
- [ ] Track Active Item in Solution Explorer
- [✓] Show advanced build configurations
- [✓] Always show solution
- [✓] Save new projects when created
- [✓] Warn user when the project location is not trusted
- [✓] Show Output window when build starts
- [ ] Prompt for symbolic renaming when renaming files

Fig 6.6 Template Path Locations

3. Open Visual Studio or Visual C# Express Edition.
4. From the menu, select **File**->**New**->**Project…**

5. The New Project dialog appears; Click on the Visual C# projects on the left side of the New Project dialog box. You should see a new template under My Templates called SJJ_MF Console Application



Basic .NET MF Console Application

Name: SPI_LCD

Location: C:\dotNETMF\Applications\App_Dev\Chapter5

Solution Name: SPI_LCD

[✓] Create directory for solution

Fig 6.7 - Including the SJJ Visual Studio Template

6. Click on **SJJ_MF Console Application.**
7. In the Name at the bottom, type in **SPI_LCD.**
8. Click **OK** to create the project. When you open Program.cs you will notice that the basic structure of the application is already setup for you. You only need to add the hardware provider for your platform.
9. From the menu, click on **Project**, and then select **Add Reference…** from the drop down menu items.
10. Click on the Browse Tab, and locate the **SJJ_Hardware provider.DLL**.
11. Click **OK**.
12. Open Program.cs.
13. Add the following to the *using* list:

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

14. Save the project

6.3.4 Adding the Code to send data to the SPI LCD

Now that we have the basic application set up, we need to configure some SPI settings and create an instance of the SPI port. Then we need to add the code to the write a string to the SPI LCD.

1. Before the Run method in the App class add the following:

```
static SPI.Configuration mySPIPortSettings = new SPI.Configuration(Pins.GPIO_NONE,
false, 0, 0, false, true, 300, SPI.SPI_module.SPI1);
SPI mySPIPort = new SPI(mySPIPortSettings);
```

The first line sets up our SPI port communication settings. Per our schematic, there is no ORing of GPIO line to create the chip-select, so GPIO_NONE is set for the chip select. The second argument sets the chip select active state, so false is required for the negative going chip select of the SPI LCD controller. We are not using a GPIO chip select OR'd with the SPI frame select signal, so the next two parameters, chip select setup time and chip select hold time are not used and are set to 0's. The 5th argument is the clock idle state. Since the clock idle state is 0, we set this parameter to false. Next is the clock edge that the data is read on, and that is the positive clock edge, so we set that parameter to true. Next is the clock frequency in KHz which is set to 300 KHz. The last parameter is the actual SPI module itself, and the iPac-9302 only has one supported. Using the configuration settings, the next line creates a new instance of the SPI port.

2. The first step is to send the basic commands to write our string to the display. As we learned early there are several byte commands that need to be sent. In the Run method add the following:

```
// LCD send byte string to lines preamble
byte[] bWriteLCDLine1Preamble = new byte[] { 0xF5, 0x8A, 0x01, 0x01, 0xF5, 0x80 };
```

The above line creates a byte array with the basic commands to set the cursor position and execute the command to write the string. This information needs to precede the string we are sending each time. The commands can be found in the user manual for the display.

3. Now, add the following line, which is our string to be displayed.

```
// Unicode literal strings
string sDisplayString1 = "SJJ EMS Welcomes to .NET MF";
```

4. In order for the string to be sent, we need to convert the Unicode string to a byte array and then merge the two byte arrays together so they can be sent. Remember that all strings in .NET MF are UTF-8 Unicode, so we will use some encoding to get the bytes. Add the following lines of code:

```
// Convert string literals to byte arrays
byte[] bDisplayString1 = new byte[sDisplayString1.Length];

// convert literal unicode strings to byte arrays
System.Text.UTF8Encoding Encoding = new System.Text.UTF8Encoding();
bDisplayString1 = Encoding.GetBytes(sDisplayString1);

// Construct the final message byte array
byte[] bWriteMessageLine1 = new byte[bWriteLCDLine1Preamble.Length +
bDisplayString1.Length];

//copy the preamble for line 1 first followed by the string1 starting at the
//end of the pre-able (using the length)
bWriteLCDLine1Preamble.CopyTo(bWriteMessageLine1, 0);
bDisplayString1.CopyTo(bWriteMessageLine1, bWriteLCDLine1Preamble.Length);
```

5. Finally write the message to the SPI LCD:

```
Thread.Sleep(1000);
mySPIPort.Write(bWriteMessageLine1);
Thread.Sleep(6);
```

6. Save the project

Here is the entire code listing:

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

namespace SPI_LCD
{
    public class Program
    {
        public static void Main()
        {
            Debug.Print(Resources.GetString(Resources.StringResources.String1));
            App myApp = new App();
            myApp.Run();
        }

        public class App
        {
            static SPI.Configuration mySPIPortSettings = new
SPI.Configuration(Pins.GPIO_NONE, false, 0, 0, false, true, 300,
SPI.SPI_module.SPI1);
            SPI mySPIPort = new SPI(mySPIPortSettings);

            public void Run()
            {

                // LCD send byte string to lines preamble
                byte[] bWriteLCDLine1Preamble = new byte[] { 0xF5, 0x8A, 0x01,
0x01, 0xF5, 0x80 };
                // Unicode literal strings
                string sDisplayString1 = "SJJ EMS Welcomes to .NET MF";

                // Convert string literals to byte arrays
                byte[] bDisplayString1 = new byte[sDisplayString1.Length];

                // convert literal unicode strings to byte arrays
                System.Text.UTF8Encoding Encoding = new
System.Text.UTF8Encoding();
                bDisplayString1 = Encoding.GetBytes(sDisplayString1);

                // Construct the final message byte array
                byte[] bWriteMessageLine1 = new byte[bWriteLCDLine1Preamble.Length
+ bDisplayString1.Length];

                //copy the preamble for line 1 first followed by the string1
starting at the
                //end of the pre-able (using the length)
                bWriteLCDLine1Preamble.CopyTo(bWriteMessageLine1, 0);
                bDisplayString1.CopyTo(bWriteMessageLine1,
bWriteLCDLine1Preamble.Length);

                Thread.Sleep(1000);
                mySPIPort.Write(bWriteMessageLine1);
                Thread.Sleep(6);


            }


        }

    }
}
```

6.3.5 Configure Project Properties, Build, and Deploy the application.

1. Make sure the Null modem cable is connected to the iPac-9302 and the development computer.
2. With the project selected in the Solution Explorer, from the menu, select **Project** and then select **SPI_LCD Properties…** from the submenu.
3. The project properties page appears, click on the **Micro Framework** tab.
4. Under Deployment, select **Serial** for the Transport and **Com#** as the Device. Where # is the COM port number of the development computer.
5. Save the project
6. From the menu, select **Build**, and then select **Build SPI_LCD** from the drop down menu.
7. Make sure the iPac-9302 is powered on. You need to wait at least 20 seconds before deploying an application. This gives TinyBooter a chance to turn control over to the CLR. You can also monitor the boot-up of the iPac-9302 with MFDeploy and use the clearing of the BootLoader Flag to shorten the boot time, but **remember to disconnect MFDeploy before you attempt to deploy an application from Visual Studio.**
8. From the menu, select **Build**, and then select **Deploy SPI_LCD**.
9. The output window should show a successful download. Power down the board, wait one second, and power up the board. The LCD should display the message: "SJJ EMS Welcomes to .NET MF". The string is longer than 16 characters so it wraps around to line 2.

***Note**: It is assumed that the LCD will turn on when power is applied and display an internal Welcome message:*

Modtronix Serial
LCD Display

thus no commands are need to enable the SPI LCD. You can check this out ahead of time by applying 5V power to the LCD2S-162. If the display appears blank, you may have to add a mySPIPort.Write({0xF5, 0x1a}) to turn on the LCD before send the display string.

6.4 Exercise 6.2: Managed Code Driver Library - Serial LCD Display

Most of the work for talking to the SPI LCD display was setting up the command to write out the data. If you have to do this every time you want to write to the display, the application would become huge and unmanageable. Also, there are commands to scroll the display, shift the display, enable/disable cursors, etc. To make writing future SPI LCD applications simpler, we will create a class driver library with all of the commands. This exercise is broken into two parts, first create the class driver library and second creating an application that uses the class driver library.

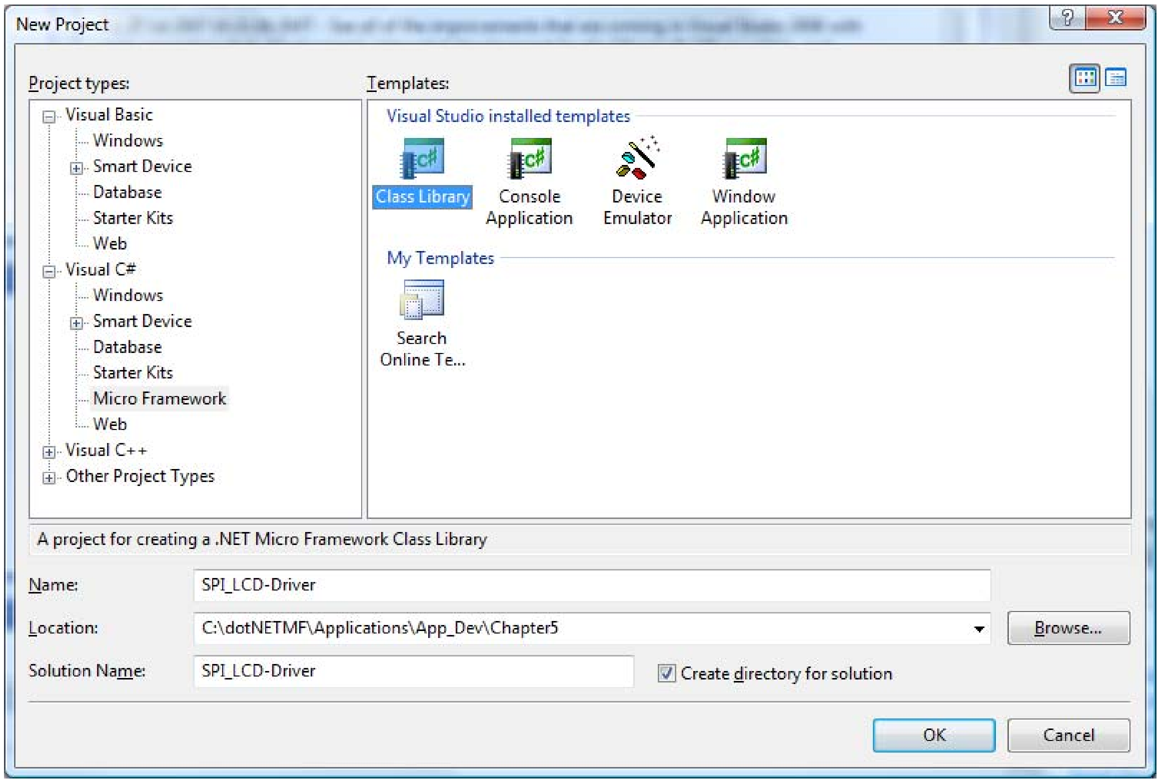
***Note:** If you open the project that is supplied with the EDK, due to possible directory differences, you will have to remove and re-add the SJJ_HardwareProvider resource.*

6.4.1 Create the SPI_LCD Driver Library

Creating a Class Library is very similar to creating any project in Visual Studio.

1. Open Visual Studio or Visual C# Express Edition.
2. From the menu, select **File**->**New**->**Project…**
3. The New Project dialog appears; expand the Visual C# projects

4. Click on **Micro Framework**
5. Select **Class Library**



A project for creating a .NET Micro Framework Class Library

Name: SPI_LCD-Driver

Location: C:\dotNETMF\Applications\App_Dev\Chapter5

Solution Name: SPI_LCD-Driver

☑ Create directory for solution

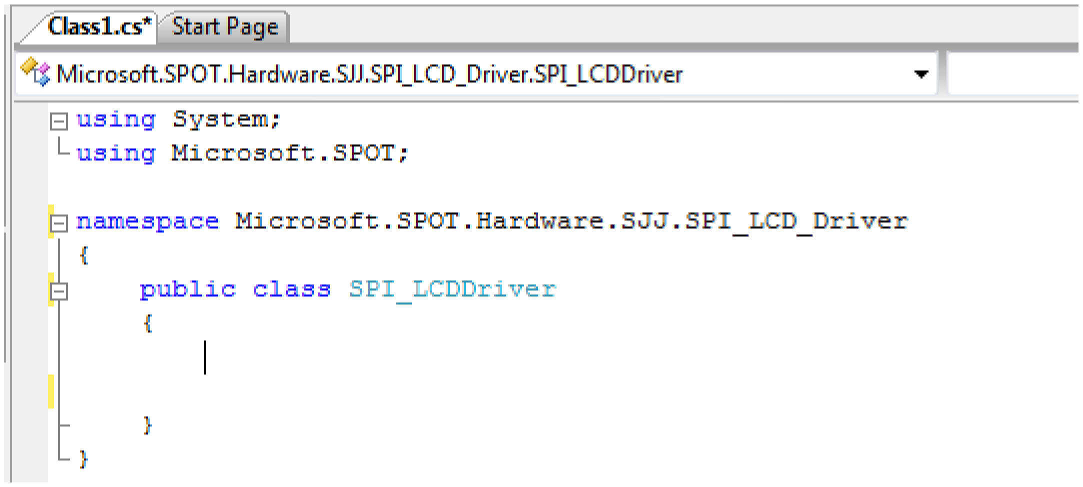
Fig 6.8 - Creating a Class Library

6. In the Name at the bottom, type in **SPI_LCD_Driver.**
7. Click **OK** to create the project.
8. With the project selected in the Solution Explorer, from the menu, click on **Project**, and then select **Add Reference…** from the drop down menu items.
9. Add the Microsoft.SPOT.Hardware
10. Click **OK**.
11. Save the project

6.4.2 Adding the code to the library

1. First thing is to change the name space. Rename the namespace from SPI_LCD-Driver to Microsoft.SPOT.Hardware.SJJ.SPI_LCD_Driver. This is how all the name spaces get created.

2. Next, rename the Class1 to SPI_LCDDriver.



```
using System;
using Microsoft.SPOT;

namespace Microsoft.SPOT.Hardware.SJJ.SPI_LCD_Driver
{
    public class SPI_LCDDriver
    {

    }
}
```

Fig 6.9 - Creating the SPI LCD Class

3. Rename the Class1.cs file to SPI_LCDDriver.cs

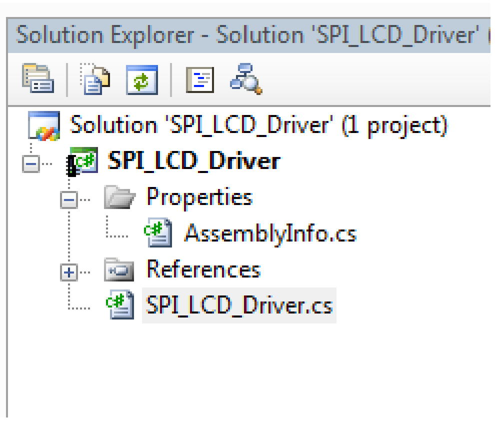

Solution Explorer - Solution 'SPI_LCD_Driver'
Solution 'SPI_LCD_Driver' (1 project)
SPI_LCD_Driver
Properties
AssemblyInfo.cs
References
SPI_LCD_Driver.cs

Fig 6.10 - Saving the SPI LCD Class Driver Source File

4. In the SPI_LCDDriver class add the following lines of code for the basic LCD commands:

```
private const int ilineLength = 16;

//SPI Keypad Reads
private byte[] SPIWriteKeyPadBuf = new byte[3] { 0xF5, 0xD1, 0x00 };
public byte[] KeyPadReadBuf = new byte[4];


// LCD Commands
// Turn display on/off
private byte[] bWriteLCDOn = new byte[] { 0xF5, 0x1A };
private byte[] bWriteLCDOff = new byte[] { 0xF5, 0x12 };
// Backlight on/off
private byte[] bWriteLCDBacklightOn = new byte[] { 0xF5, 0x28 };
private byte[] bWriteLCDBacklightOff = new byte[] { 0xF5, 0x20 };
//Cursor operations
private byte[] bWriteLCDHomeCursor = new byte[] { 0xF5, 0x8B };
private byte[] bWriteLCDBlinkblockCursorOn = new byte[] { 0xF5, 0x18 };
private byte[] bWriteLCDBlinkblockCursorOff = new byte[] { 0xF5, 0x10 };
//private byte[] bWriteLCDCusorMoveForward = new byte[] { 0xF5, 0x09 };
//private byte[] bWriteLCDCusorMoveBackward = new byte[] { 0xF5, 0x01 };
private byte[] bWriteLCDUnderlineCursorOn = new byte[] { 0xF5, 0x19 };
private byte[] bWriteLCDUnderlineCursorOff = new byte[] { 0xF5, 0x11 };
private byte[] bWriteLCDMoveCursorRight = new byte[] { 0xF5, 0x83 };
private byte[] bWriteLCDMoveCursorLeft = new byte[] { 0xF5, 0x84 };
//Display operations
private byte[] bWriteLCDShiftDisplayRight = new byte[] { 0xF5, 0x85 };
private byte[] bWriteLCDShiftDisplayLeft = new byte[] { 0xF5, 0x86 };
private byte[] bWriteLCDShiftDisplayUp = new byte[] { 0xF5, 0x87 };
private byte[] bWriteLCDShiftDisplayDown = new byte[] { 0xF5, 0x88 };
private byte[] bWriteLCDClear = new byte[] { 0xF5, 0x8C };

//LCD Read Status
private byte[] SPIWriteLCDReadStatus = new byte[3] { 0xF5, 0xD0, 0x00 };
public byte[] ReadLCDStatusBuf = new byte[4];

// LCD send byte string to lines preamble
private byte[] bWriteLCDLine1Preamble = new byte[] { 0xF5, 0x8A, 0x01, 0x01, 0xF5,
0x80 };
private byte[] bWriteLCDLine2Preamble = new byte[] { 0xF5, 0x8A, 0x02, 0x01, 0xF5,
0x80 };
```

5. Add the public methods to perform the different operations. The methods will be used in the test application.

```
public byte[] LCDOn()
{
    return (bWriteLCDOn);
}

public byte[] LCDOff()
{
    return (bWriteLCDOff);
}

public byte[] LCDBacklightOn()
{
    return (bWriteLCDBacklightOn);
}

public byte[] LCDBacklightOff()
{
    return (bWriteLCDBacklightOff);
}

public byte[] LCDHomeCursor()
{
    return (bWriteLCDHomeCursor);
}

public byte[] LCDBlinkblockCursorOn()
{
    return (bWriteLCDBlinkblockCursorOn);
}

public byte[] LCDBlinkblockCursorOff()
{
    return (bWriteLCDBlinkblockCursorOff);
}

public byte[] LCDUnderlineCursorOn()
{
    return (bWriteLCDUnderlineCursorOn);
}

public byte[] LCDUnderlineCursorOf()
{
    return (bWriteLCDUnderlineCursorOff);
}

public byte[] LCDMoveCursorRight()
{
    return (bWriteLCDMoveCursorRight);
}
```

```
public byte[] LCDMoveCursorLeft()
{
    return (bWriteLCDMoveCursorLeft);
}

public byte[] LCDClear()
{
    return (bWriteLCDClear);
}

public byte[] LCDShiftDisplayRight()
{
    return (bWriteLCDShiftDisplayRight);
}

public byte[] LCDShiftDisplayLeft()
{
    return (bWriteLCDShiftDisplayLeft);
}

public byte[] LCDShiftDisplayUp()
{
    return (bWriteLCDShiftDisplayUp);
}

public byte[] LCDShiftDisplayDown()
{
    return (bWriteLCDShiftDisplayDown);
}

public byte[] LCDReadStatus()
{
    return (SPIWriteLCDReadStatus);
}

public byte[] LCDKeyPadRead()
{
    return (SPIWriteKeyPadBuf);
}
```

6. Add two methods that create the byte array message for either display line and returns a final byte array when the method is called.

```
public byte[] LCDLine1(string sDisplayString)
{

    // Convert string literals to byte arrays
    byte[] bDisplayString = new byte[sDisplayString.Length];

    // convert literal unicode strings to byte arrays
    System.Text.UTF8Encoding Encoding = new System.Text.UTF8Encoding();
    bDisplayString = Encoding.GetBytes(sDisplayString);

    // Construct the display lines
    //Size the array
    byte[] bWriteMessageLine = new byte[bWriteLCDLine1Preamble.Length +
    bDisplayString.Length];

    //copy the preamble for line 1 first followed by the string starting at the
    //end of the pre-able (using the length)
    bWriteLCDLine1Preamble.CopyTo(bWriteMessageLine, 0);
    bDisplayString.CopyTo(bWriteMessageLine, bWriteLCDLine1Preamble.Length);

    return bWriteMessageLine;
}
```

```
public byte[] LCDLine2(string sDisplayString)
{

    // Convert string literals to byte arrays
    byte[] bDisplayString = new byte[sDisplayString.Length];

    // convert literal unicode strings to byte arrays
    System.Text.UTF8Encoding Encoding = new System.Text.UTF8Encoding();
    bDisplayString = Encoding.GetBytes(sDisplayString);

    // Construct the display lines
    //Size the array
    byte[] bWriteMessageLine = new byte[bWriteLCDLine2Preamble.Length +
    bDisplayString.Length];

    //copy the preamble for line 1 first followed by the string starting at the
    //end of the pre-able (using the length)
    bWriteLCDLine2Preamble.CopyTo(bWriteMessageLine, 0);
    bDisplayString.CopyTo(bWriteMessageLine, bWriteLCDLine1Preamble.Length);

    return bWriteMessageLine;
}
```

7. Finally, let's add two methods that write to each line of the LCD, but don't wrap the text.

```
public byte[] LCDLine1NoWrap(string sDisplayString)
{
    string sTrimString = "";
    for (int i = 0; (i < ilineLength) && (i < sDisplayString.Length); i++)
    {
        sTrimString += sDisplayString[i];
    }

    // Convert string literals to byte arrays
    byte[] bDisplayString = new byte[ilineLength];

    // convert literal unicode strings to byte arrays
    System.Text.UTF8Encoding Encoding = new System.Text.UTF8Encoding();
    bDisplayString = Encoding.GetBytes(sTrimString);

    // Construct the display lines
    //Size the array
    byte[] bWriteMessageLine = new byte[bWriteLCDLine1Preamble.Length +
    bDisplayString.Length];

    //copy the preamble for line 1 first followed by the string starting at the
    //end of the pre-able (using the length)
    bWriteLCDLine1Preamble.CopyTo(bWriteMessageLine, 0);
    bDisplayString.CopyTo(bWriteMessageLine, bWriteLCDLine1Preamble.Length);

    return bWriteMessageLine;
}

public byte[] LCDLine2NoWrap(string sDisplayString)
{
    string sTrimString = "";
    for (int i = 0; (i < ilineLength) && (i < sDisplayString.Length); i++)
    {
        sTrimString += sDisplayString[i];
    }

    // Convert string literals to byte arrays
    byte[] bDisplayString = new byte[sTrimString.Length];

    // convert literal unicode strings to byte arrays
    System.Text.UTF8Encoding Encoding = new System.Text.UTF8Encoding();
    bDisplayString = Encoding.GetBytes(sTrimString);

    // Construct the display lines
    //Size the array
    byte[] bWriteMessageLine = new byte[bWriteLCDLine2Preamble.Length +
    bDisplayString.Length];
```

```
        //copy the preamble for line 1 first followed by the string starting at the
        //end of the pre-able (using the length)
        bWriteLCDLine2Preamble.CopyTo(bWriteMessageLine, 0);
        bDisplayString.CopyTo(bWriteMessageLine, bWriteLCDLine1Preamble.Length);

        return bWriteMessageLine;
    }
```

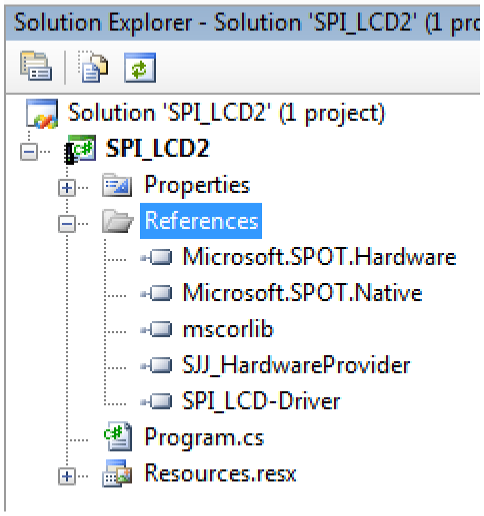
8. Save the project
9. Build both a Debug and Release version of the DLL
10. Close the project when finished. We can now use the SPI_LCD_Driver class in our applications.

6.4.3 Create a Test Application

1. Open Visual Studio or Visual C# Express Edition.
2. Using the SJJ_MF Console Application, create a new application called SPI_LCD2
3. Add the reference to the SJJ_HardwareProvider.dll
4. Open Program.cs, and add the following to the beginning of the file:

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

5. Now add our new SPI_LCD_Driver driver library to the project. You will have to browse to the \SPI_LCD-Driver\SPI_LCD-Driver\bin\Release or Debug to locate the DLL.



```
Solution Explorer - Solution 'SPI_LCD2' (1 pro
Solution 'SPI_LCD2' (1 project)
  SPI_LCD2
    Properties
    References
      Microsoft.SPOT.Hardware
      Microsoft.SPOT.Native
      mscorlib
      SJJ_HardwareProvider
      SPI_LCD-Driver
    Program.cs
    Resources.resx
```

Fig 6.11 - Adding SPI_LCD Driver Reference

6. Add the Using statement:

```
using Microsoft.SPOT.Hardware.SJJ.SPI_LCD_Driver;
```

7. Save the project.
8. Before the Run method in the App class add the following to configure the SPI port:

```
static SPI.Configuration mySPIPortSettings = new SPI.Configuration(Pins.GPIO_NONE,
false, 0, 0, false, true, 50, SPI.SPI_module.SPI1);
SPI mySPIPort = new SPI(mySPIPortSettings);
```

9. Add an instance of our LCD driver

```
SPI_LCDDriver myLCDDriver = new SPI_LCDDriver();
```

10. With the class driver all we have to do is use SPI.Writes to send commands to the LCD. In the run methods add the following:

```
//Turn backlight off
mySPIPort.Write(myLCDDriver.LCDBacklightOff());
Thread.Sleep(100);

//Clear display
mySPIPort.Write(myLCDDriver.LCDClear());
Thread.Sleep(6);

//Home cursor
mySPIPort.Write(myLCDDriver.LCDHomeCursor());
Thread.Sleep(6);

//Turn flashing cursor on
mySPIPort.Write(myLCDDriver.LCDBlinkblockCursorOn());
Thread.Sleep(6);

//Turn backlight on
mySPIPort.Write(myLCDDriver.LCDBacklightOn());
Thread.Sleep(10);

mySPIPort.Write(myLCDDriver.LCDLine1("LCD class"));
Thread.Sleep(6);

mySPIPort.Write(myLCDDriver.LCDLine2("driver is fun"));
Thread.Sleep(3000);

mySPIPort.Write(myLCDDriver.LCDShiftDisplayUp());
Thread.Sleep(6);

mySPIPort.Write(myLCDDriver.LCDLine2("Welcome to .NET MF"));
Thread.Sleep(6);

mySPIPort.Write(myLCDDriver.LCDShiftDisplayUp());
Thread.Sleep(6);

mySPIPort.Write(myLCDDriver.LCDLine2("sjjmicro.com"));
Thread.Sleep(6);
```

Note the liberal use of Thread.Sleep()'s after the write commands. This gives the LCD SPI controller a chance to process the command and guards against overrunning the SPI data buffer.

As you can see, writing to the SPI LCD is simpler when you can enumerate commands and methods in a reusable class library. You don't have to write code for the byte conversions each time you want to write to the display.

11. Change the project properties so the application can be downloaded via the Serial port.

12. Build, deploy, and run the application on the iPac-9302. The applications should first clear the display then scroll up the message.

6.5 Exercise 6.3: SPI Read - SPI Keypad Input

In the LCD driver, we added support for SPI port reads. The LCD2S-162 has support for a 4x4 keypad. In this exercise, we will perform SPI reads of the keypad buffer. You must complete the prior two exercises before starting this one.

***Note:** The LCD2S also supports a 4x3 keypad, and comes configured for that keypad from the manufacturer. To use the 4x4 keypad you will have to change a jumper and program the SPI controller on the LCD2S-162 for the different keyboard scan of 4x4.*

Items needed for this lab

- Connection Wire –9, .025" square socket to .025 square socket – www.e-z-Hook.com
- LCD2S-162 with 2x 16 (2 line x16 Character) LCD or equivalent - http://microcontrollershop.com/
- 4x4 keypad (HC-KP) - http://microcontrollershop.com/

***Note:** If you open the project that is supplied with the EDK, due to possible directory differences, you will have to remove and re-add the SJJ_HardwareProvider resource.*

6.5.1 Hardware setup

1. With the iPac-9302 powered off, wire up the SPI LCD to the iPac-9302 per section 6.3.2.
2. Next, connect the Row and Colum pins from the keypad to the Row and Colum pins on the LCD2S-162 SPI controller.
3. Connect one more cable from the keypad shield to a ground pin on the LCD2S-162.

6.5.2 Creating the SPI Keypad application

1. Open Visual Studio or Visual C# Express Edition.
2. Using the SJJ_MF Console Application, create a new application called Keypad_SPI.
3. Click **OK** to create the project.
4. With the project selected in the Solution Explorer, from the menu, click on **Project**, and then select **Add Reference…** from the drop down menu items.
5. Click on the Browse Tab, and locate the **SJJ_Hardware provider.DLL**
6. Click **OK**.
7. Click on the Browse Tab, and locate the **SPI_LCD-Driver.DLL**
8. Click **OK**.
9. Save the Project.
10. Add the following *using* declarations:

```
using Microsoft.SPOT.Hardware.SJJ;
using Microsoft.SPOT.Hardware.SJJ.SPI_LCD_Driver;
```

11. In the App class add the following to define and configure the SPI port and create an instance of the LCD driver:

```
static SPI.Configuration mySPIPortSettings = new SPI.Configuration(Pins.GPIO_NONE,
false, 0, 0, false, true, 50, SPI.SPI_module.SPI1);
SPI mySPIPort = new SPI(mySPIPortSettings);
SPI_LCDDriver myLCDDriver = new SPI_LCDDriver();
```

12. In the Run method add the following code to setup the LCD display

```
//Clear display
mySPIPort.Write(myLCDDriver.LCDClear());
Thread.Sleep(6);
```

```
//Home cursor
mySPIPort.Write(myLCDDriver.LCDHomeCursor());
Thread.Sleep(6);

//Turn flashing cursor on
mySPIPort.Write(myLCDDriver.LCDBlinkblockCursorOn());
Thread.Sleep(6);

mySPIPort.Write(myLCDDriver.LCDLine1("Keypad Test"));
Thread.Sleep(1500);


//Clear display
mySPIPort.Write(myLCDDriver.LCDClear());
Thread.Sleep(6);
```

13. Since we are reading byte codes from the keypad we need to setup a byte array with the first two members setting up the LCD to display text.

```
//Setup to write the keypad character to LCD
byte[] bWriteLCDChar = new byte[3];
bWriteLCDChar[0] = 0xF5;
bWriteLCDChar[1] = 0x80;
```

14. Finally, add the while-loop that reads the keypad buffer and writes the corresponding value to the LCD display. We take advantage of the LCD driver's support for the KeyPadRead read method and the available KeyPadReadBuff, which will hold the data.

```
while (true)
{

    mySPIPort.WriteRead(myLCDDriver.LCDKeyPadRead(), myLCDDriver.KeyPadReadBuf);
    Thread.Sleep(6);

     if ((myLCDDriver.KeyPadReadBuf[3] >= 0x61) && (myLCDDriver.KeyPadReadBuf[3]
    <= 0x70))
    {
        switch (myLCDDriver.KeyPadReadBuf[3])
        //switch(SPIReadKeyPadByteBuf[0])
        {
            case 0x61:
            {
                bWriteLCDChar[2] = 0x31;  //1
                break;
            }
            case 0x62:
            {
                bWriteLCDChar[2] = 0x32; //2
                break;
            }
            case 0x63:
            {
                bWriteLCDChar[2] = 0x33;  //3
                break;
            }
            case 0x64:
            {
                bWriteLCDChar[2] = 0xC4;    //Enter
                break;
            }
            case 0x65:
            {
                bWriteLCDChar[2] = 0x34;  //4
                break;
            }
```

```
                case 0x66:
                {
                    bWriteLCDChar[2] = 0x35; //5
                    break;
                }
                case 0x67:
                {
                    bWriteLCDChar[2] = 0x36;  //6
                    break;
                }
                case 0x68:
                {
                    bWriteLCDChar[2] = 0xC5;    //Up arrow
                    break;
                }
                case 0x69:
                {
                    bWriteLCDChar[2] = 0x37;  //7
                    break;
                }
                case 0x6A:
                {
                    bWriteLCDChar[2] = 0x38;  //8
                    break;
                }
                case 0x6B:
                {
                    bWriteLCDChar[2] = 0x39; //9
                    break;
                }
                case 0x6C:
                {
                    bWriteLCDChar[2] = 0xC6;    //Down arrow
                    break;
                }
                case 0x6D:
                {
                    bWriteLCDChar[2] = 0xB7;    //Cancel
                    break;
                }
                case 0x6E:
                {
                    bWriteLCDChar[2] = 0x30; //0
                    break;
                }
                case 0x6F:
                {
                    bWriteLCDChar[2] = 0xC8;    //Left arrow
                    break;
                }
                case 0x70:
                {
                    bWriteLCDChar[2] = 0xC7;    //Right arrow
                    break;
                }
                default:
                {
                    bWriteLCDChar[2] = 0x00;
                    break;
                }
            }

            if (bWriteLCDChar[2] != 0x00)
            {
                mySPIPort.Write(bWriteLCDChar);
            }

            Thread.Sleep(100);
        }

    }
```

The case statement converts the keypad value that the scan controller assigns to each individual key to a representative ASCII value the LCD can display. In a different application the Enter, arrow keys, and cancel would perform different functions instead of displaying to the LCD, such as branching off to perform different actions or cause the cursor to move.

15. Save the project.

6.5.3 Build, Deploy, and Test

1. Make sure the Null modem cable is connected to the iPac-9302 and the development computer.
2. With the project selected in the Solution Explorer, from the menu, select **Project** and then select **Keypad_SPI Properties…** from the submenu.
3. The project properties page appears, click on the **Micro Framework** tab.
4. Under Deployment, select **Serial** for the Transport and **Com#** as the Device. Where # is the COM port number of the development computer.
5. Save the project
6. From the menu, select **Build**, and then select **Build Keypad_SPI** from the drop down menu.
7. Make sure the iPac-9302 is powered on. You need to wait at least 20 seconds before deploying an application. This gives TinyBooter a chance to turn control over to the CLR. You can also monitor the boot-up of the iPac-9302 with MFDeploy and use the clearing of the BootLoader Flag to shorten the boot time, but **remember to disconnect MFDeploy before you attempt to deploy an application from Visual Studio.**
8. From the menu, select **Build**, and then select **Deploy Keypad_SPI**.
9. Power cycle the iPac-9302. The Keypad Test message will appear and then disappear. You can now press keys on the keypad and watch each character appear on the LCD.

6.6 The SPI Summary

In this chapter, we learned about the SPI port and writing managed code applications for the SPI port. SPI can support multiple devices on the SPI bus by using GPIOs and OR-gates to provide chip selects for each device. Not having a specific protocol, each SPI device is free to implement its own command and data interchange rules, but this frees the overhead of a protocol stack for faster operation.

The exercises focused on the SPI LCD device. Gathering the technical information on the SPI device is important to setup the SPI communication settings and understand the command protocol to transmit to and receive from the SPI device. There are other popular SPI devices available such as Analog-to-Digital converters, Real-Time controllers (RTC), GPIO expanders, 7-segment LED drivers, various PIC controllers, etc. You can apply the same analysis techniques to write a managed code application for any of these devices.

The exercise to create a class driver library for a device is important so you can re-use the code and make your applications simpler to write and debug. This comes in handy when you have multiple devices connected to the SPI bus, each having its own communications rules.

7 Serial Ports – COM2 (RS-232/422/485)

Serial ports have been mainstays in computing since the 1960s. Serial ports have been used to connect computers to devices like terminals, modems, printers, and other equipment. RS-232 is the most popular serial port, but there are also other standards like RS-422, RS-423, RS-449 and RS-485 that have had wide use. A fully populated iPac-9302 (EDKPlus) comes with the optional second serial port (HDR5), which supports RS-232, RS-422, and RS-485. This chapter will cover the following:

- An overview of RS-232, RS-422, and RS-485
- iPac-9302 COM2 hardware details
- .NET MF Serial Classes
- Basic RS-232 Communication Example

7.1 RS-232, RS-422, and RS-485 Overview

7.1.1 RS-232

RS-232 is one of the oldest and most popular serial communication standards. Personal computers have used RS-232 ports since the first PCs were introduced. Speed is a limiting factor for RS-232 topping out at 256K baud. Regardless of the limitation for today's hyper-fast PCs, many embedded systems use RS-232 and almost every microcontroller supports at least 1 RS-232 port.

Signal-wise, RS-232 defines voltage levels and logic ones and zeroes. Unlike SPI, voltage levels are within +/- 15 volts. A positive voltage is a logic zero and a negative voltage is logic one. Ground is at zero volts and voltages near zero volts are not valid RS-232 logic levels. The voltage levels themselves will be dependent on the power supply available. Some transmitter chips include special charge pump circuitry to produce the RS-232 signal voltages from a single, low voltage DC power source. These chips take the 0 and 1 logic levels and translate them to RS-232 signal levels. The iPac-9302 makes use of these transmitter chips so additional power supply inputs to support the RS-232 voltage levels are not needed. Depending on the transmitter drivers, the bit rate, and the capacitance of the cable, RS-232 cable lengths are typically a few feet long: 25 to 50 feet (6 to 15 meters) are typical. Low capacitance RS-232 cables can extend that range to up to 100 feet (30.5 meters).

7.1.2 RS-422

RS-422 offers a different signal scheme using balanced or differential signaling. The advantage of RS-422 over RS-232 is the potential for higher baud rates and longer cable lengths. The cabling defined by the standard is usually twisted pair with a maximum cable length of 3900 feet (1200 meters). RS-422 has been used to extend the transmission length between devices. RS-422 also supports multi-drop, where one driver can signal up to 10 receivers.

7.1.3 RS-485

RS-485 is another differential signal standard that supports cable lengths up to 4000 feet (1200m). Unlike RS-422, RS-485 supports multipoint serial communication, where up to 32 transceivers and 32 receivers can be connected to the same 2 wires. Communication is half-duplex. Signaling is based on voltage transmission with signals ranging from -7 to +12V. RS-485 has no data protocol defined, so it is up to the developer to define the protocol. This means careful designing must go into interfacing systems with RS-485 to avoid communication collisions.

EGPIO9 must be sent low to perform a RS-485 transmit. Set EGPIO9 low before performing a data transmit and then release high when finished. The SJJ_Hardware provider has a constant for the EGPIO9 port.

7.2 *iPac-9302 COM2 Port Hardware*

In order to achieve the voltage signal requirements for these serial specifications, some external line drivers are required to convert the digital signals from the CPU. The iPac-9302's COM 2 port includes the different line drivers to support RS-232, RS-422, and RS-485. Jumper JB2 is used to select between the communications standards. Only one can be used at a time. The only additional item software-wise is that EGPIO9 is used for RS-485 transmission. If you are going to use RS-485, EGPIO9 must be set and left to logical True or 1 early in the application before any RS-485 transmissions is made.

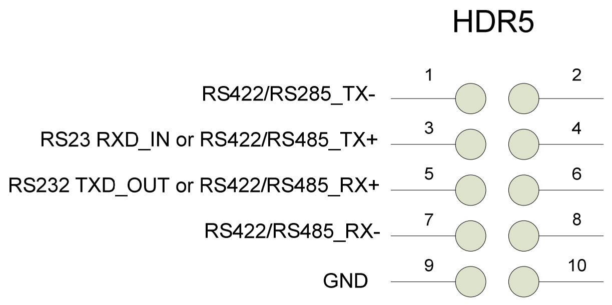

HDR5
RS422/RS285_TX-
RS23 RXD_IN or RS422/RS485_TX+
RS232 TXD_OUT or RS422/RS485_RX+
RS422/RS485_RX-
GND
1
2
3
4
5
6
7
8
9
10

Fig 7.1 - HDR5 Signals

7.3 *SerialPort Classes*

The SerialPort class has been updated from earlier versions of .NET MF to be more in line with .NET Framework 3.5 usage. There is a single class with overloaded constructors to allow ease of configuration of the SerialPort parameters for a variety of usage situations. You can specify the serial port name, the baud rate, the parity, the number of data bits, and the stop bits. The overloading lets you specify only the port name up to all parameters including the stop bits, with those parameters not be specified being set to default values.

Including the SerialPort class is a little confusing because of the naming of the .NET MF assembly that contains the support library. The name of the assembly is Microsoft.SPOT.Hardware.SerialPort. Which is a DLL: *Microsoft.SPOT.Hardware.SerialPort.DLL*. This would lead one to expect the namespace for the SerialPort class to be Microsoft.SPOT.Hardware, but it is not. The namespace for the SerialPort class is System.IO.Ports, and in this namespace is the SerialPort class. Therefore, to include the assembly library for the SerialPort class, you need to include the reference to the assembly, *Microsoft.SPOT.Hardware.SerialPort.* Then to add the namespace reference to the code, you must add the *using* statement: *using System.IO.Ports*.

The Serial class provides enumerations for the port names defined by Microsoft, but these only support "COM1" through "COM3". The EDK supports two real COM ports and 2 virtual COM ports, so port names "COM1" through "COM4" are valid names, though "COM1" is currently defaulted to the debug port and is unavailable for application access. Virtual ports "COM3" and "COM4" provide access to the PWM and ADC hardware described later. Therefore, you should use the port name enumerations provided by the SJJ Hardware Provider, *SerialPorts*. The same is true with the baud rate enumerations, the EDK supports a wider range of baud rates than the standard .NET MF enumerations provide for: 75, 150, 300, 600, 1200, 2400, 4800, 9600, 19200, 38400, 57600, & 115200 baud. Therefore, you should use the baud rate enumerations provided by the SJJ Hardware Provider, *BaudRates*. For the port names you can use the enumerations SerialPorts.COM2 through SerialPorts.COM4 (COM1 restricted to debug use); and for baud rates you can use the enumerations BaudRates.Baud75 through BaudRates.Baud115200. The enumerations provided by .NET MF for parity and stop bits may be used as is.

The SerialPort class provides the connection to the actual serial port. The basic methods are as follows:

- Open – Open access to the COM port.
- Close – Close access to the COM port & clear the buffers.
- Read – Read data from the serial port.
- Write – Write data to the serial port.
- Dispose – Releases the unmanaged serial port resources.
- Flush – Forces any data in the send buffer to be sent out and clears the send buffer.
- DiscardInBuffer – Discards the data in the receive buffer.
- DiscardOutBuffer – Discards the data in the send buffer without sending it out.

The SerialPort class provides the following properties that one can get or set:

- BaudRate – Also can be set in the constructor (get or set).
- BytesToRead – Number of bytes in the read buffer (get).
- BytesToWrite – Number of bytes in the send buffer (get).
- DataBits – Also can be set in the constructor (get or set).
- IsOpen – Indicating if the port is opened or closed (get).
- Parity – Also can be set in the constructor (get or set).
- PortName – Must be set in the constructor (get).
- ReadTimeout – Number of milliseconds before a read time-out occurs (get or set).
- StopBits – Also can be set in the constructor (get or set).
- WriteTimeout – Number of milliseconds before a write time-out occurs (get or set).
- HandShake – Enable or disable RTS or XOn/XOff handshaking for transmission (get or set).

Note: *HandShake cannot be set in the constructor. If handshaking is desired, it must explicitly be set by setting the HandShake property after the SerialPort object has been created.*

In keeping with better alignment with .NET Framework 3.5, there is now support for data receive events. This will be illustrated in the following exercise.

You still use the Serial port class to communicate using RS422 and RS485. Only the physical layer changes when you change the jumper and EGPIO9 must be true for RS485. Of course, the application needs to handle the data appropriately.

7.4 Exercise 7.1: Basic RS-232 Communication Application

The application will have the iPac-9302 communicate with SJJ_COMM (or other serial terminal application) via the COM2 port. Anything sent from the PC will be echoed back by the application.

Note: *If you open the project that is supplied with the EDK, due to possible directory differences, you will have to remove and re-add the SJJ_HardwareProvider resource.*

7.4.1 Create the Serial_Test Application

1. Open Visual Studio or Visual C# Express Edition.
2. From the menu, select **File**->**New**->**Project…**
3. The New Project dialog appears; under **Other Languages**, click on the **Visual C#** projects on the left side of the New Project dialog box. You should see a new template under My Templates called SJJ_MF Console Application.
4. Click on **SJJ_MF Console Application.**
5. In the Name at the bottom, type in **Serial_Test.**
6. Click **OK** to create the project. When you open Program.cs you will notice that the basic structure of the application is already setup for you. You only need to add the hardware provider for your platform.
7. From the menu, click on **Project**, and then select **Add Reference…** from the drop down menu items.
8. Click on the Browse Tab, and locate the **SJJ_Hardware provider.DLL**.
9. Click **OK**.
10. From the menu, click on **Project**, and then select **Add Reference…** from the drop down menu items.
11. Click on the .NET Tbo, and add the **Microsoft.SPOT.Hardware.SerialPort** reference.
12. Open Program.cs.
13. Add the following to the *using* list:

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

14. Save the project

7.4.2 Add the code to Create an instance of the serial port and Echo Characters

1. The first thing that needs to be done is to create a SerialPort object and use the constructor to configure the port settings appropriately. The SerialPort object will be for "COM2", the baud rate will be 115200, no parity, 8 data bits, and 1 stop bit. In public class App before RUN() add the following:

```
public SerialPort COM2Port = new SerialPort(SerialPorts.COM2,
(int)BaudRates.Baud115200, Parity.None, 8);
```

The above line is very different from earlier versions of .NET MF. A SerialPort object is created with the physical port, baud rate, parity, etc. already set in one operation.

***Note:** we did not explicitly set the StopBits in the constructor, because the default is 1 and that's what we want for this configuration. Also, we are not using flow control, so we will not have to explicitly set the HandShake property as it defaults to none.*

2. Next, we will add a string to send to the output, define a byte output array, and add the GreenLED GPIO object. Since the serial port class supports data receive events, we will have the application flash the GreenLED while it waits for data to come in.

```
public static String sOutput = "Outputting to COM2\n";
public byte[] boutArray = new byte[sOutput.Length];
public OutputPort myGreenLED = new OutputPort(Pins.GREEN_LED, false);
```

3. Now let's enter the code for main body of the application. First, we need to set up one more serial port property and then open the COM port. In the Run() method, add the following code:

```
COM2Port.ReadTimeout = 10;
COM2Port.Open();
```

4. Next, we want to create the DataReceive event handler. The event must be created after the opening of the COM port. The setup is like a GPIO event handler. Start typing the following:

```
COM2Port.DataReceived +=
```

   Intellisense will create a pop-up suggestion. Hit **TAB** key twice to create the COM2Port_DataReceived handler.

5. Next, what we want to do is send out our sOutput string. The string has to first be encoding using UTF8Encoding. Once completed, we need to convert each byte into our byte array, and then we can send it out the COM port and toggle the Green LED. Enter the following:

```
System.Text.UTF8Encoding Encoding = new System.Text.UTF8Encoding();
boutArray = Encoding.GetBytes(sOutput);

COM2Port.Write(boutArray, 0, boutArray.Length);
Thread.Sleep(500);

//Stay in loop to continue to receive data
while (true)
{
     Thread.Sleep(500);
     myGreenLED.Write(true);

    Thread.Sleep(500);
    myGreenLED.Write(false);
}
```

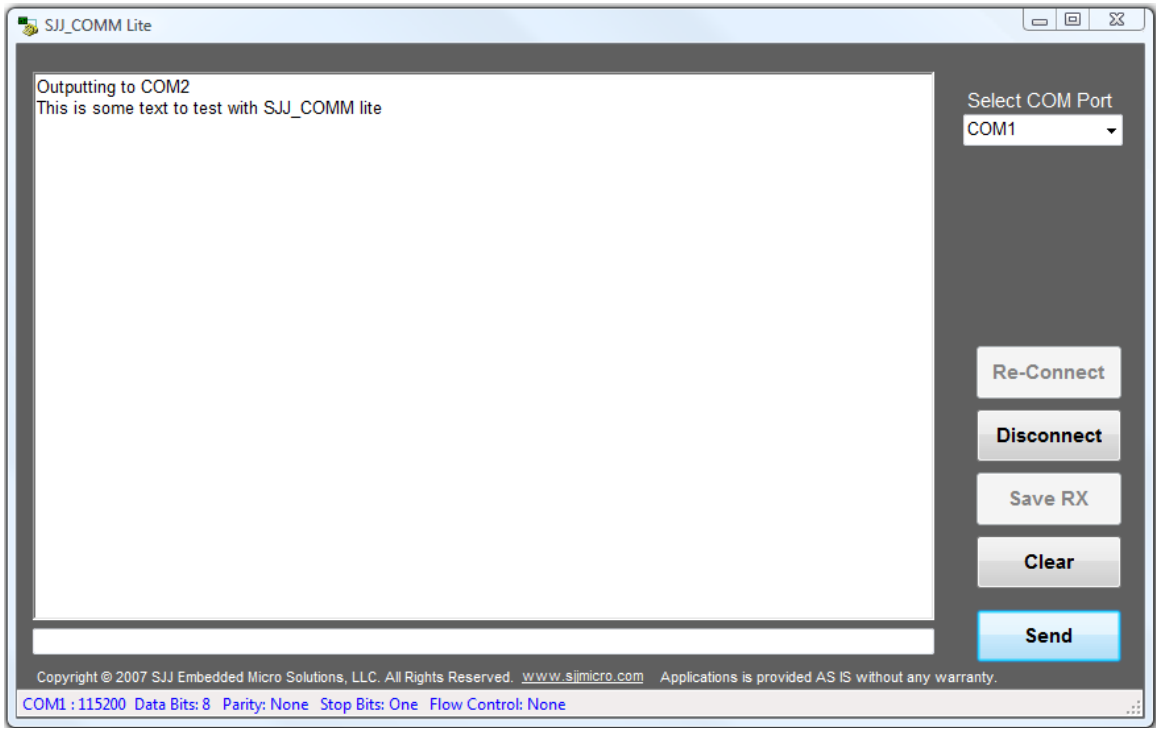
6. Finally, we need to add the code to the COM2Port_DataReceived handler. The code will simply read the data from the serial buffer and echo the data back out the COM port. In the COM2Port_DataReceived handler add the following:

```
int incount;
int iBytesToRead;

if ((iBytesToRead = COM2Port.BytesToRead) > 0)
{
    byte[] binArray = new byte[iBytesToRead];

    incount = COM2Port.Read(binArray, 0, iBytesToRead);
    COM2Port.Write(binArray, 0, incount);

}
```

When the data event occurs, the amount of data to read is not fixed. Data can continue to fill the buffer. An overflow of the binArray could occur if you use separate reads of COM2Port.BytesToRead to setup the binary array size and then set the number of bytes to read from the COM port. The iBytesToRead variable insures that the size of the input array and the number of bytes read into it, are always the same.

7.4.3 Configure Project Properties, Build, and Deploy the application

1. Make sure the Null modem cable is connected to the debug serial port of the iPac-9302 and the development computer.
2. With the **Serial_Test** project selected in the Solution Explorer, from the menu, select **Project** and then select **Serial_Test Properties…** from the submenu.
3. The project properties page appears, click on the **Micro Framework** on the left tab.
4. Under Deployment, select **Serial** for the Transport and **Com#** as the Device. Where # is the COM port number of the development computer.
5. Save the project
6. From the menu, select **Build**, and then select **Build Serial_Test** from the drop down menu. There should be no build errors
7. Make sure the iPac-9302 is powered on. You need to wait at least 20 seconds before deploying an application. This gives TinyBooter a chance to turn control over to the CLR. You can also monitor the boot-up of the iPac-9302 with MFDeploy and use the clearing of the BootLoader Flag to shorten the boot time, but **remember to disconnect MFDeploy before you attempt to deploy an application from Visual Studio.**
8. From the menu, select **Build**, and then select **Deploy Serial_Test**.
9. The output window should show a successful build and download. Power down the board. If you only have one NULL modem cable then unplug the cable from the debug serial port and connect it to the COM2 port of the iPac-9302.
10. Start SJJ_COMM Lite or the serial terminal software of your choice. Setup the terminal application parameters for a baud rate of 115200. SJJ_COMM Lite is already set for 115200. If you are using some other terminal software set the COM port parameters to 115200 baud, 8 data bits, no parity, 1 stop bit, no flow control, and no local echo.
11. Reconnect power to the iPac-9302. After the system boots up, you should see the sign on string on the display. Type something to send to the platform, and the data should be echoed back by the program. Observe that the green LED is flashing when no data is being handled by COM2.



```
Outputting to COM2
This is some text to test with SJJ_COMM lite
```

Copyright © 2007 SJJ Embedded Micro Solutions, LLC. All Rights Reserved. www.sjjmicro.com Applications is provided AS IS without any warranty.

COM1 : 115200 Data Bits: 8 Parity: None Stop Bits: One Flow Control: None

Fig 7.2 - SJJ_COMM Lite Interacting with the Serial_Text Applicaiton

7.5 Summary: Serial Ports and More Serial Ports

RS-232 serial port communication is the most widely known serial interface since the first PCs. It is still used widely in embedded systems. The iPac-9302 allows for serial port communication using one of three selectable physical interface standards: RS-232, RS-422, and RS-485.

If you need more serial ports, a SPI-to-RS232 transceiver is an ideal solution. We just so happen to have a white paper posted to our website that describes this implementation.

http://sjjmicro.com/Docs/Articles_NETMF/SPI_Serial_NET_Micro%20Framework_v1.3.pdf

8 Pulse Width Modulation (PWM)

Pulse-width modulation is a simple, yet powerful technique for controlling an analog signal with a digital device, like a microprocessor. The simplicity is in the control electronics that switch an analog signal on or off. When an analog signal is switched from off to on and then back to off, a pulse is generated. When the analog signal is switched on and off repeatedly, a pulse train is generated. By controlling the period of time from the start of one pulse to the start of the next pulse in a pulse train, the modulation frequency can be established and controlled. By controlling the period of time that each pulse of a pulse train is on and the period of time that each pulse of a pulse train is off the duty cycle can be established and controlled. Pulse-width modulation is the creation and control of a pulse train's frequency and duty cycle. Pulse-width modulation can be used to control power to a device or to transmit information. Fig 8.1 shows some example pulse trains with different fixed and varying duty cycles.

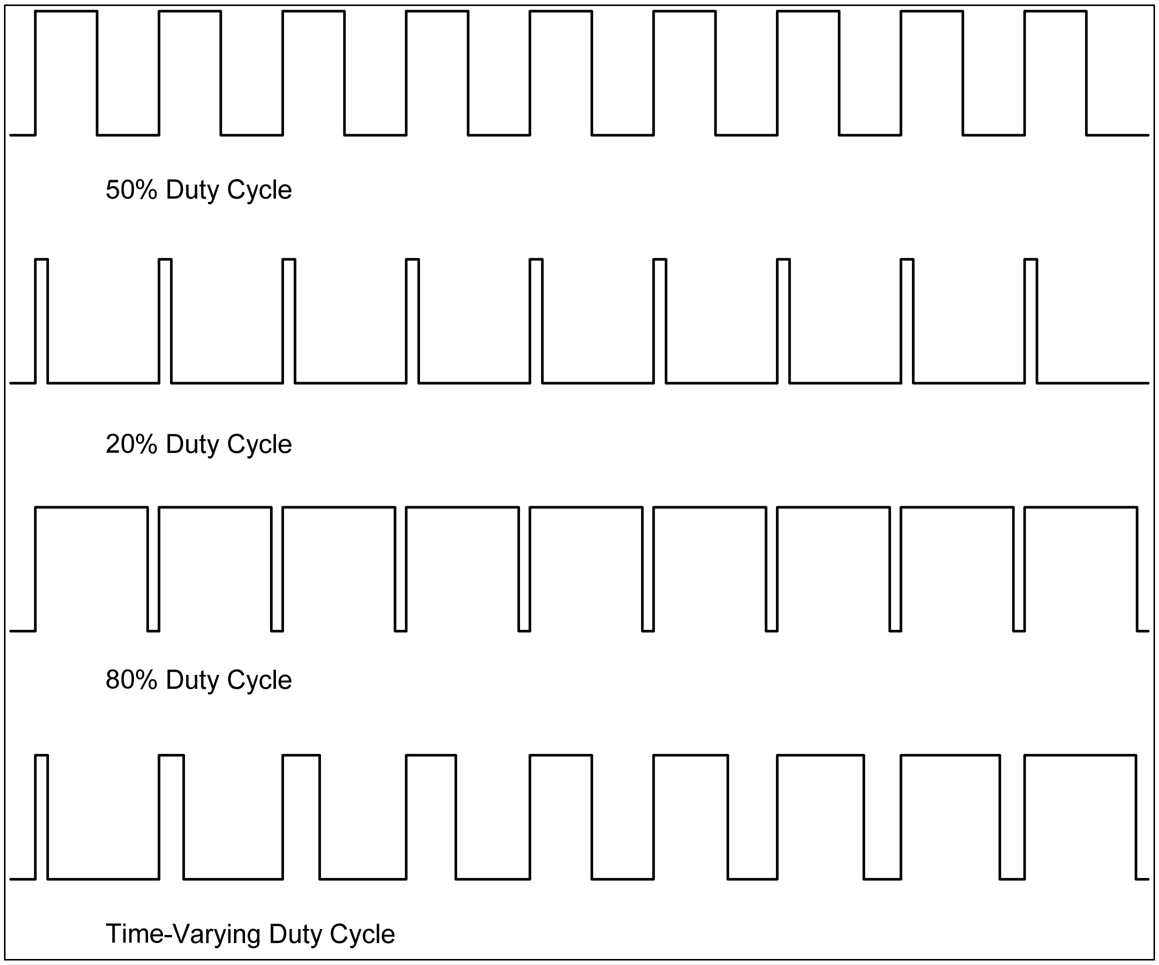


Fig 8.1 PWM Duty Cycles

8.1 *Pulse-Width Modulation Overview*

The iPac-9302 has 3 PWM's, but they are not totally independent. PWM1 (See Table D.2 - Analog (HDR2), Pin 20) is the internal PWM on the Cirrus EP9302 SOC. PLD_PWM2 (See Table D.2 - Analog (HDR2), Pin 15) and PLD_PWM3 (See Table D.2 - Analog (HDR2), Pin16) are two lower speed PWM controllers that are on the iPac-9302's CPLD. The 2 CPLD PWM controllers use the output from PWM1 as their reference frequency.

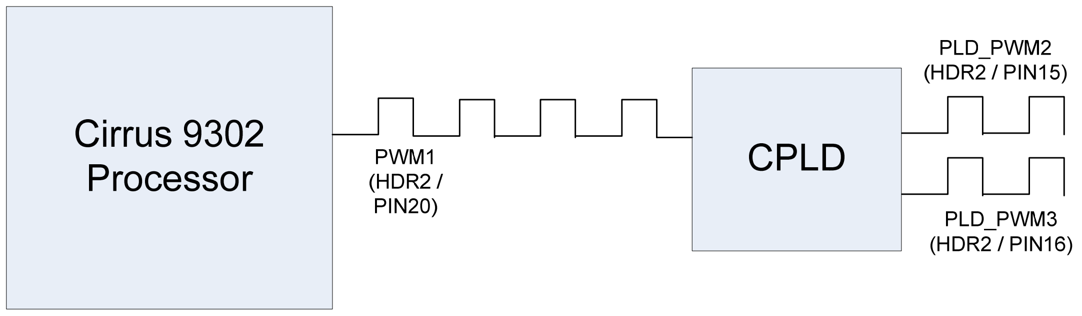


Fig 8.2 CPU PWM Provides the Reference Clock for PLD PWMS

8.1.1 Internal SOC PWM Controller

Depending on how PWM1 is programmed, it can produce a signal output from a continuous level (100% duty cycle), to a pulse train that can approach a 0% duty cycle. The output frequency of PWM1 is derived from an internal reference frequency of 14,745,600 Hz and is divided using a 16-bit counter to produce the output frequency. Another 16-bit counter is used to determine the duty cycle of the output, and one can also invert the output. The output target frequency is determined by the Terminal Count register which can be calculated as follows:

$$\text{Terminal Count Register} = \left(\frac{Reference\ Frequency}{Target\ Frequency}\right) - 1$$

The output duty cycle is determined by the Duty Cycle register which can be calculated as follows:

$$\text{Duty Cycle Register} = \left(\frac{Duty\ Cycle\ \%}{100} * (Terminal\ Count\ Register + 1)\right) - 1$$

PWM1 can be set to an output frequency from 225 Hz to 14,745,600 Hz with a duty cycle from nearly 0% to 100%.

There is also and Invert Register. This changes the steady state of the PWM1 output when the PWM is disabled and changes the phase of the switching of the output. The default state of PWM1 is for the output to be at logical 0 when the PWM is disabled. When programmed and enabled, the output will switch from logical 0 to logical 1 and remain at logical 1 for the duty cycle percentage that was programmed. Then the output will be switched to logical 0 and remain there for the duration of the remainder of the pulse train period determined by the frequency that was programmed. If the Invert Register is set to the invert state, the steady state of PWM1 will be logical 1 when the PWM is disabled. When PWM1 is programmed and enabled, it will switch from logical 1 to logical 0 and remain at logical 0 for the duty cycle percentage that was programmed. Then the output will be switched to logical 1 and remain there for the duration of the remainder of the pulse train period determined by the frequency that was programmed. When PWM1 is disabled, the output will return to the steady state condition determined by the Invert Register until it is programmed and enabled, again.

8.1.2 CPLD PWM Controllers

The CPLD PWM controllers are lower speed, lower resolution controllers that are duty cycle-only controllers. Each of these controllers has an 8-bit counter that always provides an output frequency that is equal to the input reference frequency—provided by PWM1—divided by 256. Typically one would either use PWM1 for a single PWM controller application or the two CPLD

PWMs for a two-controller application. When the CPLD PWM controllers are used, one typically sets PWM1 to the desired reference frequency with a 50% duty cycle.

Each CPLD PWM controller uses the same reference frequency provided by PWM1, but each controller has its own 8-bit counter for independent duty cycle control. The duty cycle for either of the controllers can be calculated as follows:

$$\text{Duty Cycle Register} = \left(\frac{Duty\ Cycle\ \%}{100}\right) * 256$$

Since the largest value for the 8-bit Duty Cycle Register is 255, this means that the highest duty cycle percentage is 99.6%. To produce a 100% duty cycle, the Invert Control needs to be used.

As with PWM1, PLD_PWM2 and PLD_PWM3 each have Invert Control bits that can reverse the steady state output level and the phase of the switching.

8.2 PWM Programmatic Access

The .NET Micro Framework CLR does not directly support a PWM API. To provide programmatic access to the PWM controllers, we have chosen to implement a virtual COM port interface. COM3 has been configured to manage the PWM controllers. The way this works is that you must open the COM3 serial port as you would an actual serial port. Then you write a series of bytes that are interpreted at the hardware driver level as specific commands for the PWM controllers. If data is requested to be returned, then the write is followed by a read to retrieve the requested data.

It is certainly possible for you to manage the PWM's by directly using the virtual COM port and the command set that the PWM driver implements, but we have further simplified this by providing a managed code PWM driver that encapsulates all the COM port manipulations and the command definitions, and provides a more straightforward API that is more intuitive for controlling the PWM's. The managed code driver is call SJJ_PWM_Driver and is provided as a DLL that can be included as a reference when creating your managed code application.

For those of you who want to get down and dirty, there is a more detail of the inner workings of the PWM driver in Appendix B.

8.2.1 PWM Managed Code Driver API

The SJJ_PWM_Driver provides a public class called SJJ_PWM_Driver in the Microsoft.SPOT.Hardware.SJJ namespace. This class exposes the following basic driver methods to open, configure, and control the PWM controllers:

```
public SerialPort PWM_Open();
```

This method opens the virtual COM port for the PWM controllers. It returns the SerialPort object which must be stored and passed to other control methods.

```
public SJJ_PWM_Driver.PWM_Status PWM_SetFreq(uint uiPWMFreq, SerialPort
myPwm);
```

This method accepts the desired integer frequency and the SerialPort object. It calculates the appropriate register value for PWM1 Terminal Count Register and writes that value to the Terminal Count register. Only PWM1 has frequency control, so if you are using CPLD PWM's make sure that the PWM1 frequency that you select is 256X the desired CPLD PWM frequency. If the method is successful, it will return a status of eStatusWriteSuccess.

```
public SJJ_PWM_Driver.PWM_Status
PWM_SetDutyCycle(SJJ_PWM_Driver.PWM_Channel ePWMChannel, int
iDutyCycPerCent, SerialPort myPwm);
```

This method accepts the PWM channel number, which identifies which PWM controller you are configuring, the desired duty cycle as an integer percent, and the SerialPort object. It calculates the appropriate duty cycle register value for the particular PWM controller specified and writes it to that controller's duty cycle register. If the method is successful, it will return a status of eStatusWriteSuccess.

```
public SJJ_PWM_Driver.PWM_Status
PWM_SetInvert(SJJ_PWM_Driver.PWM_Channel ePWMChannel, bool bInvert,
SerialPort myPwm);
```

This method accepts the PWM channel number, which identifies which PWM controller you are configuring, a Boolean for the desired invert state, and the SerialPort object. It sets the appropriate PWM controller's invert state based on the Boolean state: true = invert on; false = invert off. . If the method is successful, it will return a status of eStatusWriteSuccess.

```
public SJJ_PWM_Driver.PWM_Status PWM_On(SerialPort myPwm);
```

This method accepts the SerialPort object and turns on the PWM1 controller. The CPLD PWM controllers do not have independent on/off control. If the method is successful, it will return a status of eStatusWriteSuccess.

```
public SJJ_PWM_Driver.PWM_Status PWM_Off(SerialPort myPwm);
```

This method accepts the SerialPort object and turns off the PWM1 controller. The CPLD PWM controllers do not have independent on/off control. If the method is successful, it will return a status of eStatusWriteSuccess.

The driver also provides some more advanced driver methods for reading the PWM1 registers and general PWM status:

```
public ushort PWM_ReadTerminalCount(SerialPort myPwm);
```

This method accepts the SerialPort object and returns the 16-bit contents of the PWM1 Terminal Count register. Only the PWM1's registers are readable.

```
public ushort PWM_ReadDutyCyle(SerialPort myPwm);
```

This method accepts the SerialPort object and returns the 16-bit contents of the PWM1 Duty Cycle register. Only the PWM1's registers are readable.

```
public SJJ_PWM_Driver.PWM_Status PWM_ReadStatus(SerialPort myPwm);
```

This method accepts the SerialPort object and returns the virtual COM port driver status.

The driver class also provides a number of enumerated types to manage the status return and identify the particular PWM controller:

```
public enum PWM_Status
{
    eStatusInit = 0,
    eStatusWriteSuccess = 1,
    eStatusReadSuccess = 2,
    eStatusReadFail = 3,
    eStatusSynchError = 4,
    eStatusCommandLengthError = 5,
    eStatusSynchSuccess = 6,
    eStatusSetReadSuccess = 7,
    eStatusSetReadFail = 8,
    eStatusOpenSuccess = 9,
    eStatusOpenFail = 10,
    eStatusCloseSuccess = 11,
    eStatusCloseFail = 12,
    eStatusPWMComHandleError = 13,
    eNumStatus = 14,
}

public enum PWM_Channel
{
    ePWM1 = 0,
    ePWM2_CPLD = 1,
    ePWM3_CPLD = 2,
}
```

Using these methods, and enumerated constants, it is a very simple task to open, configure, start, and stop the PWM controllers.

8.3 Exercise 8.1: Basic PWM Control Application

This application will configure the iPac-9302's PWM controllers using PWM1 as a 500K Hz clock source for the CPLD PWM's and then will set up PLD_PWM2 for a duty cycle of 25% and PLD_PWM3 for a duty cycle of 75% initially. Then we will swap the duty cycle settings between the two CPLD PWMs back and forth in a loop. We will also flash the green LED on and off as we iterate through the loop as a healthy status indicator.

***Note**: If you open the project that is supplied with the EDK, due to possible directory differences, you will have to remove and re-add the SJJ_HardwareProvider and SJJ_PWM_Driver resources.*

8.3.1 Create the PWM_Test Application

1. Open Visual Studio or Visual C# Express Edition.
2. From the menu, select **File→New→Project…**
3. The New Project dialog appears; under **Other Languages**, click on the **Visual C#** projects on the left side of the New Project dialog box. You should see the SJJ template under My Templates called SJJ_MF Console Application.
4. Click on **SJJ_MF Console Application.**
5. In the Name: at the bottom of the New Project dialog, type in **PWM_Test**

6. Click **OK** to create the project. Open Program.cs and, again, you will see the basic structure of the application is setup for you. You only need to add the hardware provider for your platform and the PWM managed code driver.
7. From the menu, click on **Project**, and then select **Add Reference…** from the drop down menu items.
8. Click on the **Browse** Tab, and locate the **SJJ_Hardware provider.DLL**.
9. Click **OK**.
10. From the menu, again, click on **Project**, and then select **Add Reference…** from the drop down menu items.
11. Click on the **Browse** Tab, and locate the **SJJ_PWM_Driver.dll**.
12. Click **OK**.
13. Finally, click on **Project**, and then select **Add Reference…** from the drop down menu items. In the .NET Tab, add the **Microsoft.SPOT.Hardware.SerialPort** reference.
14. Open Program.cs.
15. Add the following to the *using* list:

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

16. Save the project

8.3.2 Add the code to create the PWM driver object and green LED GPIO object

1. First, let's create the PWM driver object and the green LED object. In public class App before RUN(), add the following:

```
// Create PWM driver object
SJJ_PWM_Driver myPWMDriver = new SJJ_PWM_Driver();

// Create green LED object
OutputPort myGreenLED;
```

2. Now, let's create the PWM driver instance and the green LED GPIO instance. In the public void Run() method add the following:

```
// Open the PWM device
SerialPort myPwmHandle = myPWMDriver.PWM_Open();

// Create green LED object for visual feedback and turn it off to start
myGreenLED = new OutputPort(Pins.GREEN_LED, false);
```

3. We need a status variable to check the result of the driver methods, and we need a Boolean variable to keep track of the duty cycle state as we iterate through the work loop. So, add the following:

```
SJJ_PWM_Driver.PWM_Status eStatusPWM;
bool bToggle = false;
```

4. Before we add our work loop, let's make sure the PWM1 is off and configured for a frequency of 500 K Hz and a duty cycle of 50% as the reference clock for the CPLD PWM's. To do this, add the following:

```
// Turn PWM1 off
eStatusPWM = myPWMDriver.PWM_Off(myPwmHandle);
if (eStatusPWM != SJJ_PWM_Driver.PWM_Status.eStatusWriteSuccess)
{
    Debug.Print("PWM_Off failed: " + eStatusPWM.ToString());
}

// Set PWM1 frequency to 500K Hz
eStatusPWM = myPWMDriver.PWM_SetFreq(500000, myPwmHandle);
```

```
if (eStatusPWM != SJJ_PWM_Driver.PWM_Status.eStatusWriteSuccess)
{
    Debug.Print("PWM_SetFreq failed: " + eStatusPWM.ToString());
}

// Set PWM1 duty cycle to 50%
eStatusPWM = myPWMDriver.PWM_SetDutyCycle(SJJ_PWM_Driver.PWM_Channel.ePWM1, 50,
myPwmHandle);
if (eStatusPWM != SJJ_PWM_Driver.PWM_Status.eStatusWriteSuccess)
{
    Debug.Print("PWM_SetDutyCycle ePWM1 failed: " + eStatusPWM.ToString());
}
```

5. Now, let's set the initial CPLD PWM duty cycles to be 25% for CPLD PWM2 and 75% for CPLD PWM3. To do this, add the following:

```
// Set CPLD PWM2 duty cycle to 25%
eStatusPWM = myPWMDriver.PWM_SetDutyCycle(SJJ_PWM_Driver.PWM_Channel.ePWM2_CPLD,
25, myPwmHandle);
if (eStatusPWM != SJJ_PWM_Driver.PWM_Status.eStatusWriteSuccess)
{
    Debug.Print("PWM_SetDutyCycle ePWM2_CPLD failed: " + eStatusPWM.ToString());
}

// Set CPLD PWM3 duty cycle to 75%
eStatusPWM = myPWMDriver.PWM_SetDutyCycle(SJJ_PWM_Driver.PWM_Channel.ePWM3_CPLD,
75, myPwmHandle);
if (eStatusPWM != SJJ_PWM_Driver.PWM_Status.eStatusWriteSuccess)
{
    Debug.Print("PWM_SetDutyCycle ePWM3_CPLD failed: " + eStatusPWM.ToString());
}
```

6. The final thing to do, before we enter the work loop, is to turn the PWMs on, light the green LED for status, and then delay so we can observe the initial results of our setup. To accomplish this, add the following:

```
// Turn on PWM
eStatusPWM = myPWMDriver.PWM_On(myPwmHandle);
if (eStatusPWM != SJJ_PWM_Driver.PWM_Status.eStatusWriteSuccess)
{
    Debug.Print("PWM_On failed: " + eStatusPWM.ToString());
}

// Turn on green LED
myGreenLED.Write(true);

Thread.Sleep(500);
```

7. Now, we are ready to add our work loop, so add the following:

```
// Loop and toggle CPLD duty cycles
while (true)
{
}
```

8. We will now add code to the work loop that looks at the toggle Boolean variable and depending on its state will change the CPLD duty cycles and then toggle the state of the Boolean. As a healthy runtime status indicator, we will also toggle the state of the green LED each time we toggle the duty cycles. Add the following to the work loop:

```
if (bToggle)
{
    bToggle = false;

    // Set CPLD PWM2 duty cycle to 75%
    eStatusPWM =
myPWMDriver.PWM_SetDutyCycle(SJJ_PWM_Driver.PWM_Channel.ePWM2_CPLD, 75,
myPwmHandle);
```

```
            if (eStatusPWM != SJJ_PWM_Driver.PWM_Status.eStatusWriteSuccess)
            {
                Debug.Print("PWM_SetDutyCycle ePWM2_CPLD failed: " +
        eStatusPWM.ToString());
            }

            // Set CPLD PWM3 duty cycle to 25%
            eStatusPWM =
        myPWMDriver.PWM_SetDutyCycle(SJJ_PWM_Driver.PWM_Channel.ePWM3_CPLD, 25,
        myPwmHandle);
            if (eStatusPWM != SJJ_PWM_Driver.PWM_Status.eStatusWriteSuccess)
            {
                Debug.Print("PWM_SetDutyCycle ePWM3_CPLD failed: " +
        eStatusPWM.ToString());
            }

            // Turn off green LED
            myGreenLED.Write(false);

        }
        else
        {
            bToggle = true;

            // Set CPLD PWM2 duty cycle to 25%
            eStatusPWM =
        myPWMDriver.PWM_SetDutyCycle(SJJ_PWM_Driver.PWM_Channel.ePWM2_CPLD, 25,
        myPwmHandle);
            if (eStatusPWM != SJJ_PWM_Driver.PWM_Status.eStatusWriteSuccess)
            {
                Debug.Print("PWM_SetDutyCycle ePWM2_CPLD failed: " +
        eStatusPWM.ToString());
            }

            // Set CPLD PWM3 duty cycle to 75%
            eStatusPWM =
        myPWMDriver.PWM_SetDutyCycle(SJJ_PWM_Driver.PWM_Channel.ePWM3_CPLD, 75,
        myPwmHandle);
            if (eStatusPWM != SJJ_PWM_Driver.PWM_Status.eStatusWriteSuccess)
            {
                Debug.Print("PWM_SetDutyCycle ePWM3_CPLD failed: " +
        eStatusPWM.ToString());
            }

            // Turn on green LED
            myGreenLED.Write(true);

        }

        Thread.Sleep(500);
```

We added and arbitrary delay at the bottom of the loop to give us some time to observe the duty cycle changes. It should also be noted, that we do not turn off the PWMs before we change the duty cycles and then turn them back on. One would do this if you were concerned about pulse width jitter while the duty cycle registers are being written. In this simple case, we do not care about transition jitter.

8.3.3 Configure Project Properties, Build, and Deploy the application

1. Make sure the Null modem cable is connected to the debug serial port of the iPac-9302 and the development computer.
2. With the **PWM_Test** project selected in the Solution Explorer, from the menu, select **Project** and then select **PWM_Test Properties…** from the submenu.
3. The project properties page appears, click on the **Micro Framework** on the left tab.
4. Under Deployment, select **Serial** for the Transport and **Com#** as the Device. Where # is the COM port number of the development computer.
5. Save the project

6. From the menu, select **Build**, and then select **Build PWM_Test** from the drop down menu. There should be no build errors
7. Make sure the iPac-9302 is powered on. You need to wait at least 20 seconds before deploying an application. This gives TinyBooter a chance to turn control over to the CLR. You can also monitor the boot-up of the iPac-9302 with MFDeploy and use the clearing of the BootLoader Flag to shorten the boot time, but **remember to disconnect MFDeploy before you attempt to deploy an application from Visual Studio.**
8. From the menu, select **Build**, and then select **Deploy PWM_Test**.

The output window should show a successful download. Power down the board.

8.3.4 Running PWM_Test

Repower the iPac-9302 and when the application runs, you should see the green LED flashing, showing that the programming is running successfully. If you attach a scope to the PWM outputs, you should see the following conditions alternate with the flashing of the green LED (orange trace1 is PWM1, cyan trace 2 is CPLD_PWM2, and magenta trace 3 is CPLD_PWM3):

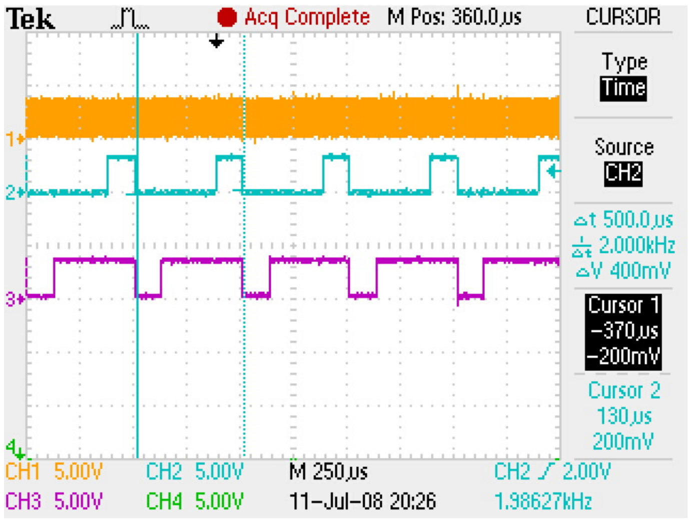

Tek
Acq Complete M Pos: 360.0μs
CURSOR
Type
Time
Source
CH2
Δt 500.0μs
1/Δt 2.000kHz
ΔV 400mV
Cursor 1
−370μs
−200mV
Cursor 2
130μs
200mV
CH1 5.00V CH2 5.00V M 250μs CH2 ╱ 2.00V
CH3 5.00V CH4 5.00V 11−Jul−08 20:26 1.98627kHz

Fig 8.3 CPLD PWM2 25% Duty Cycle; CPLD PWM3 75% Duty Cycle

Note that the frequency of the CPLD PWMs is 2K Hz as expected; 500K Hz / 256.

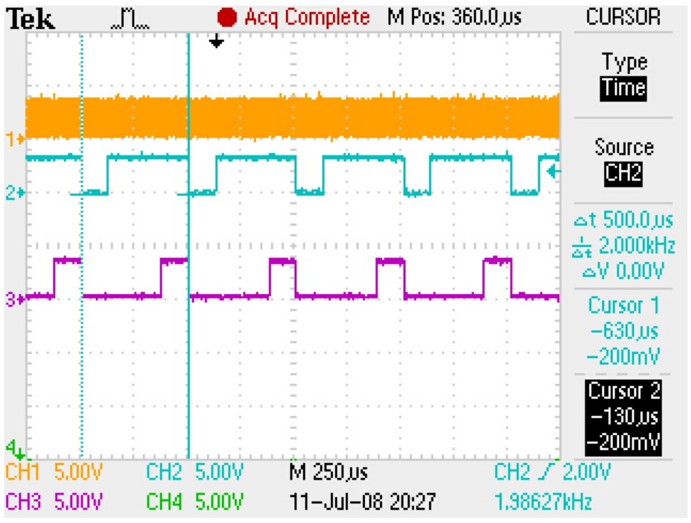

Tek
Acq Complete M Pos: 360.0μs
CURSOR
Type
Time
Source
CH2
Δt 500.0μs
1/Δt 2.000kHz
ΔV 0.00V
Cursor 1
−630μs
−200mV
Cursor 2
−130μs
−200mV
CH1 5.00V CH2 5.00V M 250μs CH2 ╱ 2.00V
CH3 5.00V CH4 5.00V 11−Jul−08 20:27 1.98627kHz

Fig 8.4 CPLD PWM2 75% Duty Cycle; CPLD PWM3 25% Duty Cycle

If we increase the sweep speed on the scope we can capture PWM1, we can verify that it is running at about the 500K Hz that we set with a 50% duty cycle.

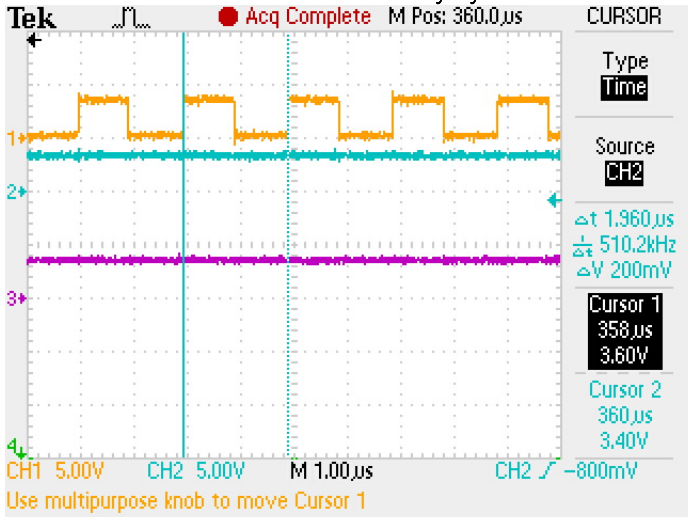


Fig 8.5 PWM1 Reference Frequency with 50% Duty Cycle

8.4 SUMMARY: Pulse Width Modulation (PWM)

The iPac-9302 has 3 PWM's. PWM1 is the internal PWM on the Cirrus EP9302 SOC and the clock source for the two CPLD PWMs. PLD_PWM2 and PLD_PWM3 are two lower speed PWM controllers that are on the iPac-9302's CPLD. The PWMs can be set to duty cycles ranging from nearly 0% to 100% with PWM1 having 1-bit duty cycle resolution and PWM2 and PWM3 having 8-bit resolution. The output frequency of the CPLD PWMs is the PWM1 frequency divided by 256. All PWMs also have an invert function that reverses the off state and the phase of output switching.

The .NET Micro Framework CLR does not directly support a PWM API. To provide programmatic access to the PWM controllers, a virtual COM port interface has been implemented. COM3 has been configured to manage the PWM controllers. A managed code PWM driver that encapsulates all the COM port manipulations and the command definitions, and provides a more straightforward API that is more intuitive for controlling the PWMs, has been provided. The managed code driver, SJJ_PWM_Driver, is provided as a DLL that can be included as a reference when creating your managed code application. The inner workings of the PWM driver are discussed in Appendix B.

9 Analog-To-Digital Converter (ADC)

An analog-to-digital converter, or ADC, is an electronic device that is used to convert an analog voltage signal, sampled at a particular instance, into a digital value. An analog voltage signal may assume any value within a continuous range, as opposed to a digital voltage signal, which will be one of two values: logical 0 or logical 1. Most of the input that we sense as humans is in analog form, i.e. sound, light, pressure, or temperature to name a few. Analog voltage signals that most electronic devices processes are typically a voltage representation of the input sources just mentioned. Analog signals of light, sound, pressure, temperature, etc. are converted to a corresponding analog voltage using a transducer. In order to process these analog voltage signals with digital electronics, like microprocessors, it is necessary to convert the time-varying analog voltage from a transducer or other source into a time-varying series of digital measurements. The ADC accomplishes just that. The ADC periodically samples one or more analog signals and converts each sample of each signal into a digitally coded representation of each analog signal at the time it was sampled. The digital coding format and the range of digital values will vary depending on the design of the ADC.

Typical encoding schemes are single-ended binary with the results being a positive binary value that is linearly related to the analog signal and ranging from some minimum value to some maximum value or bipolar binary with the results being a signed binary value that is linearly related to the analog signal and ranging from some maximum negative value to some maximum positive value. ADC digital encoding is by no means limited to these basic examples. BCD encoding, Gray code, or other digital encoding schemes can be used and the digital conversion does not have to be linear. Logarithmic, parabolic, or other non-linear conversion schemes can be implemented as well.

A typical ADC transfer function is shown in Fig 9.1.

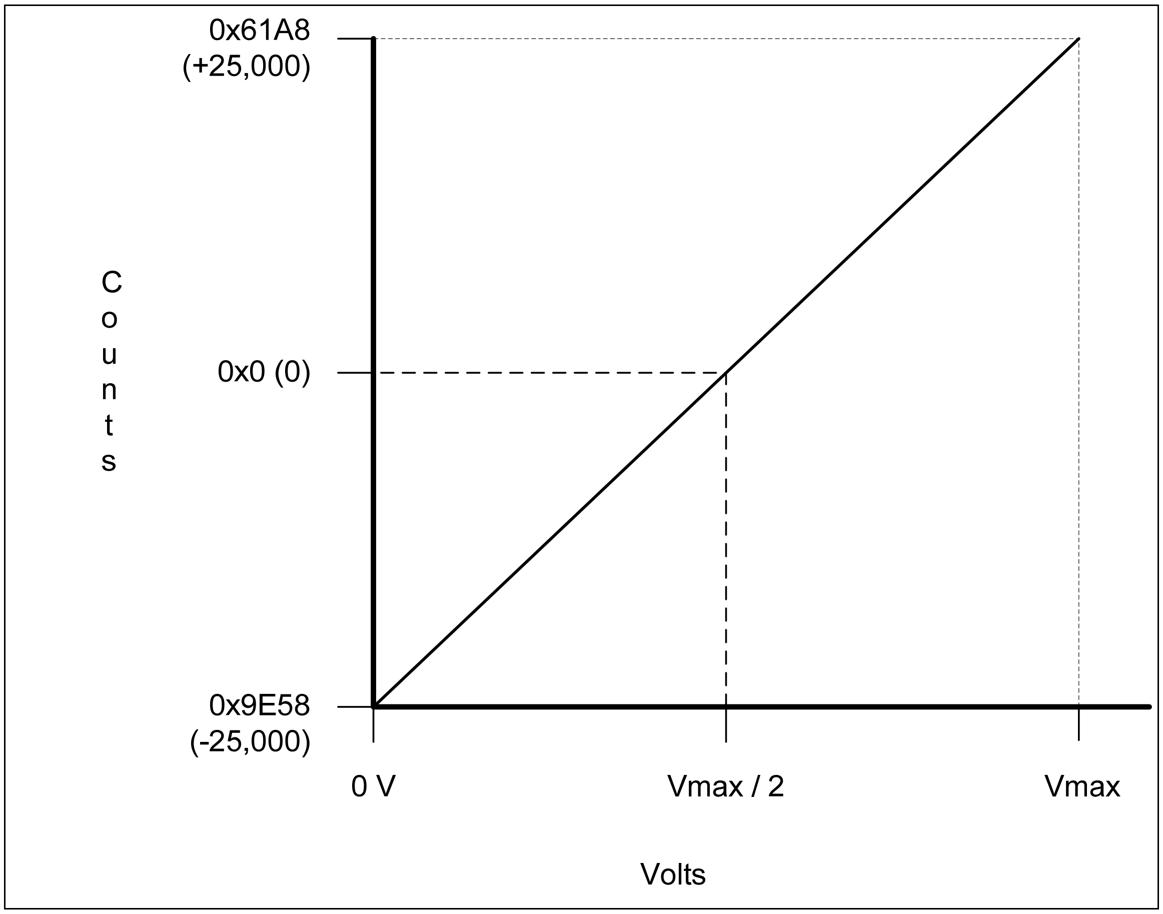


Fig 9.1 ADC Typical Transfer Function

9.1 Analog-To-Digital Converter Overview

The iPac-9302 has a single, multi-function, multi-input ADC controller which is contained in the Cirrus EP9302 SOC. The EP9302's ADC can be configured to interface to a variety of different types of touch screens or it can be configured as a 6-channel general ADC, which is how the iPac-9302 is configured. There is no support for touch screen with the iPac-9302. There are 5 channels available to monitor 5 independent analog signals, each with a voltage range of from 0 Volts to + 3.3 Volts (see Table D.2 - Analog (HDR2), ADC0, Pin1; ADC1, Pin 2, ADC2, Pin 3; ADC3, Pin4; ADC4, Pin5). The 6th channel is connected to battery voltage, VBat. The digital conversions are presented as 16-bit signed integers.

As we mentioned in the general ADC description, when an ADC is connected to an analog signal and turned on, it will takes repetitive samples of the analog signal, convert each sample to a digital value and present them to the microprocessor for processing. The iPac-9302's ADC can sample at 2 rates, which are derived from an internal clock and a 2-state divider. The ADC can sample at either 3,750 samples-per-second or 925 samples-per-second. For best system response the higher sample rate is recommended.

The sample transfer function in Fig 9.1 is, in fact, the iPac-9302's transfer function. The conversion count ranges from -25,000 counts for 0 Volts applied to the sampling channel, to 0 counts for $\frac{3.3}{2}$ Volts applied to the sampling channel, and to a maximum of +25,000 counts for 3.3 Volts applied to the sampling channel with a linear response. There will be some variation from chip-to-chip and depending on ambient temperature, so for the most accurate conversion results, you should determine a method in your application for calibrating the device. There is also a settle-time for the ADC when the input is switched from one channel to another and is dependent on the sample rate. You do not have to worry about this; because when you select the desired sample rate, the driver automatically calculates the appropriate settle time and inserts the required delay when switching channels.

9.2 ADC Programmatic Access

The .NET Micro Framework CLR does not directly support an ADC API. To provide programmatic access to the ADC controller, a virtual COM port interface has been implemented. COM4 has been configured to manage the ADC controller. The way this works, is that you must open the COM4 serial port as you would an actual serial port. Then you write a series of bytes that are interpreted at the hardware driver level as specific commands for the ADC controller. If data is requested to be returned, then the write is followed by a read to retrieve the requested data.

It is certainly possible for you to manage the ADC by directly using the virtual COM port and the command set that the ADC driver implements, but we have further simplified this by providing a managed code ADC driver that encapsulates all the COM port manipulations and the command definitions, and provides a more straightforward API that is more intuitive for controlling the ADC. The managed code driver is call SJJ_ADC_Driver and is provided as a DLL that can be included as a reference when creating your managed code application.

For those of you who want to get down and dirty, there is a more detailed description of the inner workings of the ADC driver in Appendix B.2.

9.2.1 ADC Managed Code Driver API

The SJJ_ADC_Driver provides a public class called SJJ_ADC_Driver in the Microsoft.SPOT.Hardware.SJJ namespace. This class exposes the following basic driver methods to open, configure, and control the ADC controllers:

```
public SerialPort ADC_Open();
```

This method opens the virtual COM port for the ADC controller. It returns the SerialPort object which must be stored and passed to other control methods.

```
public SJJ_ADC_Driver.ADC_Status
ADC_SetClkDiv(SJJ_ADC_Driver.ADC_ClkDiv eADCDivide, SerialPort myAdc);
```

This method accepts the clock divider enumeration and the SerialPort object. The clock divider will determine which of the sample rates the ADC will use as it make repetitive samples. If the method is successful, it will return a status of eStatusWriteSuccess.

```
public SJJ_ADC_Driver.ADC_Status
ADC_SetChannel(SJJ_ADC_Driver.ADC_Source eAnalogChannel, SerialPort
myAdc);
```

This method accepts the analog channel number and the SerialPort object. It sets the ADC controller to monitor the selected analog input and delays the appropriate settle time before returning. If the method is successful, it will return a status of eStatusWriteSuccess.

```
public SJJ_ADC_Driver.ADC_Status ADC_On(SerialPort myAdc);
```

This method accepts the SerialPort object and turns on the ADC controller. If the method is successful, it will return a status of eStatusWriteSuccess.

```
public SJJ_ADC_Driver.ADC_Status ADC_Off(SerialPort myAdc);
```

This method accepts the SerialPort object and turns off the ADC controller. If the method is successful, it will return a status of eStatusWriteSuccess.

```
public int ADC_Read(SerialPort myAdc);
```

This method accepts the SerialPort object and causes the ADC to sample the selected analog input channel. It returns the converted value as a signed integer.

```
public SJJ_ADC_Driver.ADC_Status ADC_ReadStatus(SerialPort myAdc);
```

This method accepts the SerialPort object and returns the virtual COM port driver status.

The driver class also provides a number of enumerated types to manage the status return and identify the particular ADC channel:

```
public enum ADC_Source
{
    eAnalogChannel0 = 0,
    eAnalogChannel1 = 1,
    eAnalogChannel2 = 2,
    eAnalogChannel3 = 3,
    eAnalogChannel4 = 4,
    eBatteryVoltage = 5,
}

public enum ADC_ClkDiv
{
    eADCDivide4 = 0,
    eADCDivide16 = 1,
}

public enum ADC_Status
{
    eStatusInit = 0,
    eStatusWriteSuccess = 1,
    eStatusReadSuccess = 2,
    eStatusReadFail = 3,
    eStatusSynchError = 4,
    eStatusCommandLengthError = 5,
    eStatusSynchSuccess = 6,
    eStatusSetReadSuccess = 7,
    eStatusSetReadFail = 8,
    eStatusADCSourceError = 9,
    eStatusADCComHandleError = 10,
}
```

Using these methods, and enumerated constants, it is a very simple task to open, configure, start, and stop the ADC controller.

9.3 Exercise 9.1: Basic ADC Control Application

This application will configure the iPac-9202 ADC controller to sample at the high sampling rate and then sequentially sample each of analog channels in a loop. We will also flash the green LED on and off as we iterate through the loop as a healthy status indicator.

***Note**: If you open the project that is supplied with the EDK, due to possible directory differences, you will have to remove and re-add the SJJ_HardwareProvider and SJJ_ADC_Driver resources*

9.3.1 Create the ADC_Test Application

1. Open Visual Studio or Visual C# Express Edition.
2. From the menu, select **File→New→Project…**
3. The New Project dialog appears; under **Other Languages**, click on the **Visual C#** projects on the left side of the New Project dialog box. You should see the SJJ template under My Templates called SJJ_MF Console Application.
4. Click on **SJJ_MF Console Application.**
5. In the Name: at the bottom of the New Project dialog, type in **ADC_Test**
6. Click **OK** to create the project. Open Program.cs and, again, you will see the basic structure of the application is setup for you. You only need to add the hardware provider for your platform and the ADC managed code driver.
7. From the menu, click on **Project**, and then select **Add Reference…** from the drop down menu items.
8. Click on the **Browse** Tab, and locate the **SJJ_Hardware provider.DLL**.
9. Click **OK**.
10. From the menu, again, click on **Project**, and then select **Add Reference…** from the drop down menu items.
11. Click on the **Browse** Tab, and locate the **SJJ_ADC_Driver.dll**.
12. Click **OK**.
13. Finally, from the menu, click on **Project**, and then select **Add Reference…** from the drop down menu items. In the .NET Tab, add the **Microsoft.SPOT.Hardware.SerialPort** reference.
14. Open Program.cs.
15. Add the following to the *using* list:

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

16. Save the project

9.3.2 Add the code to create the ADC driver object and the green LED GPIO object.

9. First, let's create the ADC driver object and the green LED object. In public class App before RUN() add the following:

```
// Create ADC driver object
SJJ_ADC_Driver myADCDriver = new SJJ_ADC_Driver();

// Create green LED object
OutputPort myGreenLED;
```

10. Now, let's create the ADC driver instance and the green LED GPIO instance. In the public void Run() method add the following:

```
// Open the ADC device
SerialPort myAdcHandle = myADCDriver.ADC_Open();

// Create green LED object for visual feedback and turn it off to start
myGreenLED = new OutputPort(Pins.GREEN_LED, false);
```

11. We need an index variable to walk us through each of the ADC channels as we go around the loop each time, we need an integer to hold the ADC result when we read each channel, and we need a status variable to check the result of the driver methods; so add the following:

```
int iADCSourceIdx = 0;
Int16 i16ConvertSample;
SJJ_ADC_Driver.ADC_Status eStatus;
```

12. Before we add our work loop, let's set the conversion rate to the high rate by setting the low clock divisor. Add the following:

```
// Set ADC clock divider to /4
eStatus = myADCDriver.ADC_SetClkDiv(SJJ_ADC_Driver.ADC_ClkDiv.eADCDivide4,
myAdcHandle);
if (eStatus != SJJ_ADC_Driver.ADC_Status.eStatusWriteSuccess)
{
    Debug.Print("ADC_SetClkDiv failed: " + eStatus.ToString());
}
```

13. Now, we are ready to add our work loop, so add the following:

```
// Start ADC Testing
// Sample all analog input channels
while (true)
{
}
```

14. We will now add the code to manage the ADC each time we go through the loop and sample a different analog signal channel. Each time we start through the loop we want to turn the ADC off. Inside the while (true) loop, start by adding the following:

```
SJJ_ADC_Driver.ADC_Status eStatus;

// Turn off ADC
eStatus = myADCDriver.ADC_Off(myAdcHandle);
if (eStatus != SJJ_ADC_Driver.ADC_Status.eStatusWriteSuccess)
{
    Debug.Print("ADC_Off failed: " + eStatus.ToString());
}
```

15. To iterate our way through the ADC analog signal channels, we'll now set up a case statement using the `iADCSourceIdx` variable that we created earlier. We initialized it to 0 when we created it and we will increment it at the bottom of the work loop so we can sample the next channel on the next iteration through the loop. When we increment the index, we will check its value and wrap the index back to 0 when we have successfully sampled all the input channels. Add the following case statement to the work loop:

```
// Set new ADC source
switch (iADCSourceIdx)
{
    case 0:
        eStatus =
myADCDriver.ADC_SetChannel(SJJ_ADC_Driver.ADC_Source.eAnalogChannel0,
myAdcHandle);
        if (eStatus != SJJ_ADC_Driver.ADC_Status.eStatusWriteSuccess)
        {
            Debug.Print("ADC_SetChannel eAnalogChannel0 failed: " +
eStatus.ToString());
        }
        break;
    case 1:
        eStatus =
myADCDriver.ADC_SetChannel(SJJ_ADC_Driver.ADC_Source.eAnalogChannel1,
myAdcHandle);
        if (eStatus != SJJ_ADC_Driver.ADC_Status.eStatusWriteSuccess)
        {
            Debug.Print("ADC_SetChannel eAnalogChannel1 failed: " +
eStatus.ToString());
        }
        break;
    case 2:
        eStatus =
myADCDriver.ADC_SetChannel(SJJ_ADC_Driver.ADC_Source.eAnalogChannel2,
myAdcHandle);
        if (eStatus != SJJ_ADC_Driver.ADC_Status.eStatusWriteSuccess)
        {
            Debug.Print("ADC_SetChannel eAnalogChannel2 failed: " +
eStatus.ToString());
        }
        break;
    case 3:
        eStatus =
myADCDriver.ADC_SetChannel(SJJ_ADC_Driver.ADC_Source.eAnalogChannel3,
myAdcHandle);
        if (eStatus != SJJ_ADC_Driver.ADC_Status.eStatusWriteSuccess)
        {
            Debug.Print("ADC_SetChannel eAnalogChannel3 failed: " +
eStatus.ToString());
        }
        break;
    case 4:
        eStatus =
myADCDriver.ADC_SetChannel(SJJ_ADC_Driver.ADC_Source.eAnalogChannel4,
myAdcHandle);
        if (eStatus != SJJ_ADC_Driver.ADC_Status.eStatusWriteSuccess)
        {
            Debug.Print("ADC_SetChannel eAnalogChannel4 failed: " +
eStatus.ToString());
        }
        break;
    default:
        eStatus =
myADCDriver.ADC_SetChannel(SJJ_ADC_Driver.ADC_Source.eBatteryVoltage,
myAdcHandle);
        if (eStatus != SJJ_ADC_Driver.ADC_Status.eStatusWriteSuccess)
        {
            Debug.Print("ADC_SetChannel eBatteryVoltage failed: " +
eStatus.ToString());
        }
        break;
}
```

16. Now that we have selected the analog signal channel, let's complete the loop by turning on the ADC, reading the channel, flashing the green LED, and incrementing the channel selection index. Add the following to the work loop to complete the test application:

```
Debug.Print("Source Selected");
// Turn on ADC
eStatus = myADCDriver.ADC_On(myAdcHandle);
if (eStatus != SJJ_ADC_Driver.ADC_Status.eStatusWriteSuccess)
{
    Debug.Print("ADC_On failed");
}
// Toggle green LED on at beginning of ADC sample
if ((iADCSourceIdx % 2) == 0)
{
    // Toggle green LED on
```

```
            myGreenLED.Write(true);
        }
        else
        {
            // Toggle green LED off
            myGreenLED.Write(false);
        }

        Debug.Print("ADC On");

        // Read and output ADC
        i16ConvertSample = (Int16)myADCDriver.ADC_Read(myAdcHandle);
        if ((i16ConvertSample == SJJ_ADC_Driver.ADC_COM_ERROR) || (i16ConvertSample ==
        SJJ_ADC_Driver.ADC_CONVERSION_ERROR))
        {
            Debug.Print("ADC_Read failed: " + i16ConvertSample.ToString());
        }
        else
        {
            Debug.Print("ADC Read for Ch " + iADCSourceIdx.ToString() + " = " +
        i16ConvertSample.ToString());
        }

        // Increment and wrap index
        if (++iADCSourceIdx > 5)
        {
            iADCSourceIdx = 0;
        }

        Thread.Sleep(500);
```

The loop delay time is arbitrary and set relatively long to give you a chance to read the results of each iteration through the loop in the debug pane in Visual Studio as it is happening. When you execute the program, check that the green LED is flashing as expected, as well.

9.3.3 Configure Project Properties, Build, and Deploy the application

1. Make sure the Null modem cable is connected to the debug serial port of the iPac-9302 and the development computer.
2. With the **ADC_Test** project selected in the Solution Explorer, from the menu, select **Project** and then select **ADC_Test Properties…** from the submenu.
3. The project properties page appears, click on the **Micro Framework** on the left tab.
4. Under Deployment, select **Serial** for the Transport and **Com#** as the Device. Where # is the COM port number of the development computer.
5. Save the project
6. From the menu, select **Build**, and then select **Build ADC_Test** from the drop down menu. There should be no build errors
7. Make sure the iPac-9302 is powered on. You need to wait at least 20 seconds before deploying an application. This gives TinyBooter a chance to turn control over to the CLR. You can also monitor the boot-up of the iPac-9302 with MFDeploy and use the clearing of the BootLoader Flag to shorten the boot time, but **remember to disconnect MFDeploy before you attempt to deploy an application from Visual Studio.**
8. From the menu, select **Build**, and then select **Deploy ADC_Test**.

The output window should show a successful download. Power down the board.

9.3.4 Running ADC_Test

The ADC_Test program can be executed and the debug output monitored with SJJ_COMM Lite or it can be run through the Visual Studio debugger. If you run it through the debugger, check the debug output pane in Visual Studio for the Debug.Print messages. For this test output, the ADC inputs were alternately jumper connected to ground and +3.3V.

HDR2 Pin	ADC Channel	Connection
1	ADC 0	GND
2	ADC 1	3.3V
3	ADC 2	GND
4	ADC 3	3.3V
5	ADC4	GND

Table 9.1 - ADC Connections for Exercise

***Note**: Do not connect the ADC channels to higher than 3.3 volts or the results will be invalid and damage to the ADC might occur.*

You can see the readings alternate between maximum negative value and maximum positive value. Channel 5 is VBatt which is approximately ½ of +V. Set up SJJ_COMM Lite for the same COM port as you configured the deployment port to be in Visual Studio, and you should see the following displayed in the SJJ_COMM Lite output windows:

```
        •
        •
        •
Ready.
Hello World!
Source Selected
ADC On
ADC Read for Ch 0 = -24484
Source Selected

ADC On
ADC Read for Ch 1 = 24986

Source Selected
ADC On
ADC Read for Ch 2 = -24489
Source Selected
ADC On
ADC Read for Ch 3 = 24984
Source Selected
ADC On
ADC Read for Ch 4 = -24486
Source Selected
ADC On
ADC Read for Ch 5 = 179
        •
        •
        •
```

Fig 9.2 ADC_Test Output

9.4 SUMMARY: Analog-To-Digital Converter (ADC)

An analog-to-digital converter, or ADC, is an electronic device that is used to convert an analog voltage signal, sampled at a particular instance, into a digital value. Analog voltage signals that most electronic devices process are typically a voltage representation of a physical parameter like: light, sound, pressure, temperature, etc. that are converted to a corresponding analog voltage using a transducer.

The iPac-9302 has a single, multi-function, multi-input ADC controller which is configured as a 6-channel general ADC. There are 5 channels available to monitor 5 independent analog signals, each with a voltage range of from 0 Volts to + 3.3 Volts. The 6th channel is connected to battery voltage, VBat. The digital conversions are presented as 16-bit signed integers with a range of ±25,000 counts.

The .NET Micro Framework CLR does not directly support an ADC API. To provide programmatic access to the ADC controller, a virtual COM port interface has been implemented. COM4 has been configured to manage the ADC controller. A managed code ADC driver that encapsulates all the COM port manipulations and the command definitions, and provides a more straightforward API that is more intuitive for controlling the ADC, has been provided. The managed code driver, SJJ_ADC_Driver, is provided as a DLL that can be included as a reference when creating your managed code application. The inner workings of the ADC driver are discussed in Appendix B.2.

10 Ethernet

Ethernet was developed in the 1970's by Robert Metcalfe and his assistant, David Boggs, when they were working for Xerox at the Palo Alto Research Center (PARC). Xerox was building the first laser printer and they wanted all of the computers at PARC to be able to print with this new printer. Robert Metcalfe was given the task to connect the then hundreds of computers at PARC to the laser printer with a network that would be fast enough to drive the new, very fast laser printer. In 1976 Metcalfe and Boggs published the paper: "Ethernet: Distributed Packet-Switching For Local Computer Networks" and Ethernet was born. Ethernet originally used a shared coaxial cable as the data transmission media and a frame-based communications scheme, carrier sense multiple access with collision detection (CSMA/CD) provided the means for many computers, connected to the same wire, to talk to one another. Over time, the coaxial cable was replaced with point-to-point twisted pair connections interconnected with hubs and switches.

The Ethernet standard is now defined in the IEEE 802.3 standards and has become the most popular network solution of choice, overtaking Token Ring and ARCNET. Network speeds range from 10MB/s to1GB/s. The iPac-9302 provides support for a twisted pair connection running at 10/100 MB/s.

Ethernet provides the basic physical connection and address routing for the data frames. On top of that are a number of transport and data protocols. A method for communicating data across a packet-switched network is TCP/IP, or Transmission Control Protocol/Internet Protocol. TCP/IP provides a packet addressing means to allow data to be reliably exchanged between two computers. The iPac-9302 has a TCP/IP stack built into the .NET MF CLR.

10.1 Sockets

Sockets were first implemented in the Berkley version of UNIX in the late 1980's and provide communication channels that allow unrelated processes to exchange data across networks. Sockets are independent of the network protocol, and may be hosted on TCP/IP network protocol networks. The .NET MF CLR provides a native sockets implementation that provides a standardized means for establishing a network connection and for sending and receiving data through the sockets connection. The TCP/IP and sockets implementation can be used to host both client and server application on the iPac-9302.

In .NET MF, the Sockets Class is used as the primary means for network communication in a managed code application. There are three main names spaces:

- System.Net
- System.Net.Sockets
- Microsoft.SPOT.Net.NetworkInformation

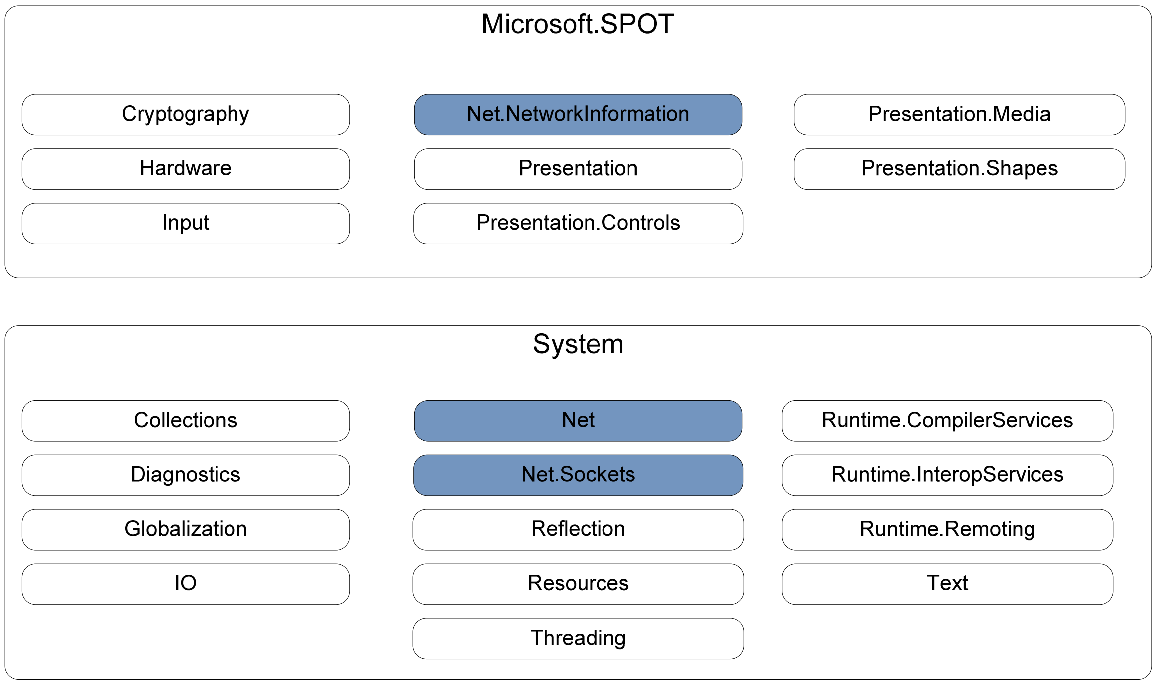


Fig 10.1 Network Namespaces

Since the networking stack was introduced in .NET MF V2.5, feature support has increased to include HTTP, HTTPs, SSL, and DPWS. Once again, the goal is to address small footprint devices, and since the iPac-9302 is headless and support for HTTP, HTTPs, SSL, and DPWS significantly increases the CLR's memory footprint, these features are not currently supported in the supplied EDK image; but they can be added upon request.

The NetworkInformation namespace provides a means to obtain network information about the device. It is similar to using IPCONFIG.exe on the desktop. It allows you to retrieve the current network settings, renew DHCP leases, set a specific static TCP/IP address, etc,

One tip as you are developing a new application is to output the network information via Debug.Print statements early in the application so you know what the network parameters are, like the IP address for example.

Just like the desktop, sockets are used for networking communications in .NET Micro Framework. The support, like other classes, is reduced to support for small systems. The SDK comes with a couple socket application examples.

The only protocol available is IPv4, and TCP and UDP are the protocol types available. Data is sent and receive using byte arrays. You will have to convert the data classes of your application to and from the byte arrays for use in send and receive operations.

HTML application pages can be accessed for parsing information. For example: an application can read data from an HTML page, and then use the information to perform certain actions. Another example is that an application can send static http strings so data could be read from a browser, but full web services are not supported in the EDK.

The .NET Micro Framework implementation of sockets is limited, so look carefully at the API reference before attempting to use code from Windows.

10.1.1 Socket Setup

An application can be setup so the system acts like a client, initiating the connection, or a server, listening for client connections. The following examples are from the two sample applications in the SDK.

The server creates a new socket and binds the socket to the address/port of the system. The final step is to listen on the port for incoming connections. Upon receipt of a connect request, the server can accept the connection and perform socket.receive and socket.send operations. All data received and sent by the server is as byte[].

```
Socket server = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);
 IPEndPoint localEndPoint = new IPEndPoint(IPAddress.Any, 12000);
 server.Bind(localEndPoint);
 server.Listen(Int32.MaxValue);
```

The client creates a new socket and then makes a connection to an external server. The client can perform a socket.send and socket.receive. All data is sent and received as byte[].

```
String Server = "www.microsoft.com";
 IPHostEntry hostEntry = Dns.GetHostEntry(server);
 Socket socket = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);
 socket.Connect(new IPEndPoint(hostEntry.AddressList[0], port));
```

10.2 Exercise 10.1 Server Access – Loopback

In this exercise, the iPac-9302 is configured as a server that will listen for a connection from a client. This is a simple example, so the server application will only support a single connection at a time. A desktop client application has been created that will establish a Sockets connection with the server application, send control messages to the server, and receive confirmation messages from the server. This document is not intended to be a desktop programming primer, so the coding details of the desktop client application will not be discussed. A description of the desktop client will follow and full source code is provided. The coding details of the .NET MF server application will be discussed.

10.2.1 The Desktop Client Application

The DesktopClient application is located in the ZIP file under Chapter 10. The application is a Windows form-based application written in Visual C#. When you run the application, you will see the following form and controls:

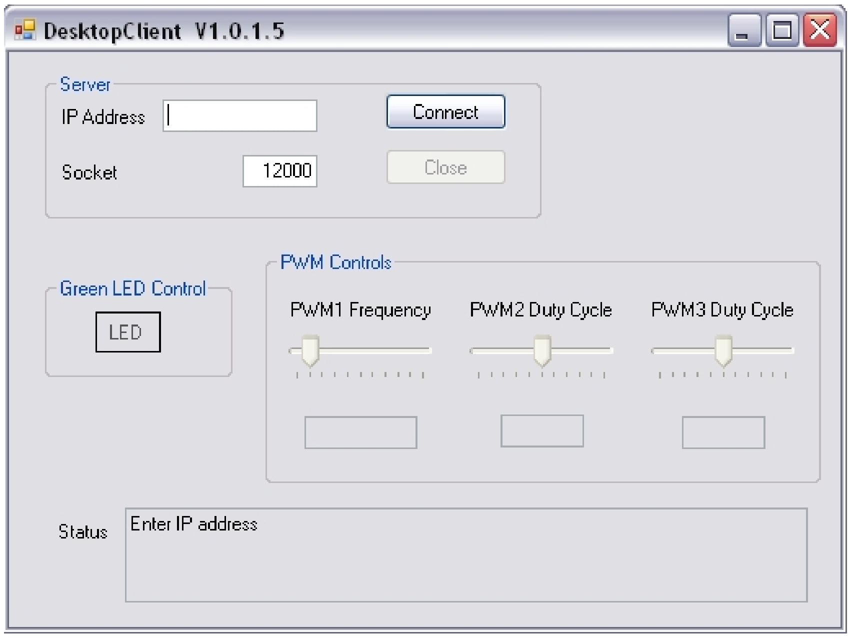


Fig 10.2 DesktopClient

The DesktopClient provides server controls that allow you to specify the IP address of the server, the socket port number that you want to communicate over, a button to request the connection and a button to close the connection. The IP address of the server can be determined in a couple of ways and will be discussed later. The default socket port number is 12000, and that is the port number that is coded into the server application. If you want to use a different port, you will need to edit the server application code and recompile it; and then enter that port number in Socket textbox of the DesktopClient.

Once the connection is made, the Green LED and PWM controls will become active. These controls give you the capability to send control commands from the DesktopClient to the server to control the .NET MF device. The LED button will send alternately an On or Off command to the server. The status of the LED button will not change until the server acts on the command and echoes the command back to the client as confirmation that the command was carried out. At this point, the color of the LED button will change to green, if the LED was commanded On, or back to background gray, if the LED was commanded Off.

The client application also has TrackBar controls for the PWMs. The first TrackBar controls the frequency of PWM1, which is used as the clock frequency for PWM2, and PWM3 (Review Section 8 Pulse Width Modulation (PWM) for PWM specifics). The client can send frequency set commands from the minimum, 225 Hz, to 100 KHz. Setting the control to zero will turn PWM1 off. The TextBox under the TrackBar control will indicate the frequency selected, bit it will not be updated instantly. After the server receives a PWM1 frequency set command and it performs the operation, it echoes back the command to the client for confirmation. When the client receives the echoed message, it will update the TextBox with the frequency set. The PWM2 and PWM3 TrackBar controls and associated TextBoxes work similarly, except they send commands to set the duty cycle of PWM2 and PWM3 respectively to a range of 0% to 100%. The TextBoxes are updated with the duty cycle set upon receipt of the echoed command.

On the bottom of the form is a multi-line status text box. This TextBox will display information about the connection, commands being send, echoed confirmation messages being received, and any error messages that might occur. When the application first launches, the status TextBox prompts you to enter the IP address of the server.

10.2.2 Socket Server Application

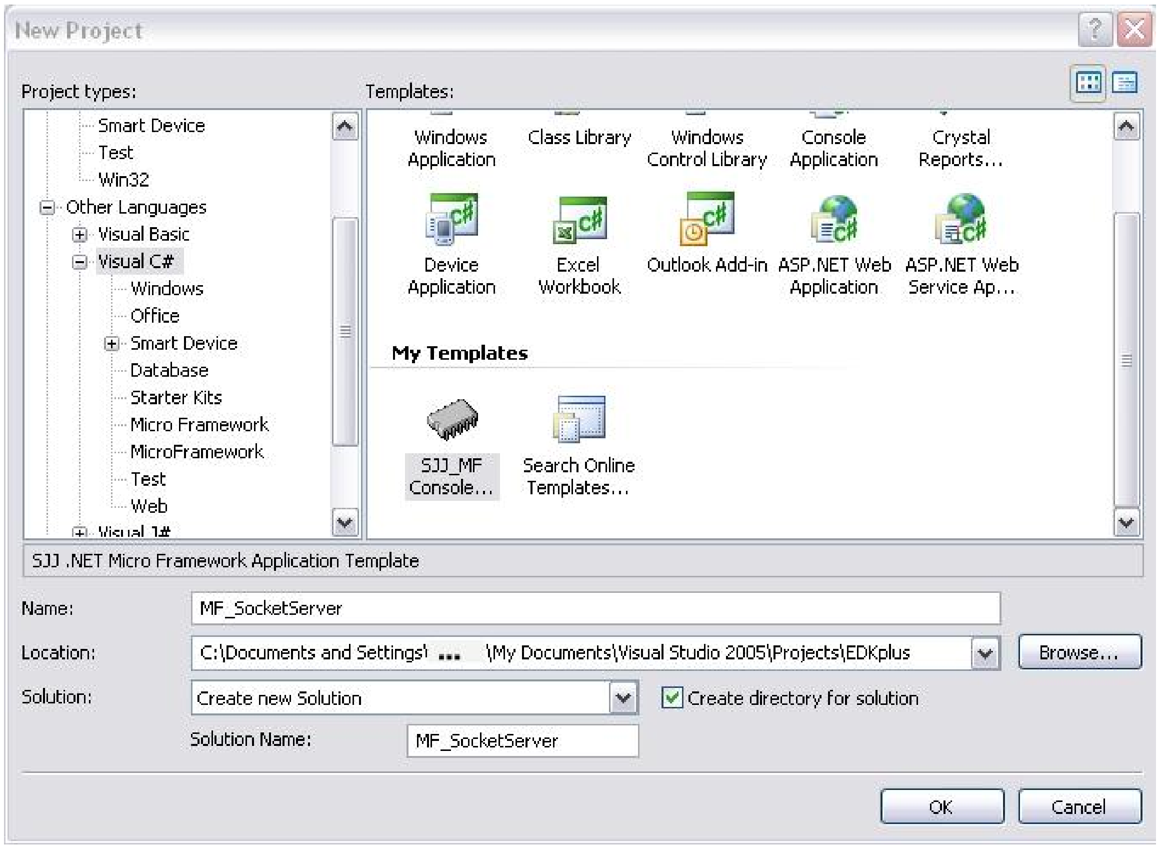
The Socket Server application is the companion application that is designed specifically to run with the Desktop Client application above. The source code to the server application is part of the examples zip file under the chapter 10 folder. When the Socket Server runs, it will sign-on through the debug port and identify itself and output the iPac-9302's IP address and the socket port number that it is listen on. The IP address that the Socket Server reports should be entered in the Desktop Client's IP address TextBox. The default port number, 12000, is the same for both applications, and does not need to be changed unless there is a conflict with some other application. To change the socket port, you need to modify the iPort constant in the App class, which will be described below, and rebuild the server application.

After the Socket Server signs on, it listens for a Sockets connection. When it gets a connection request, it will establish the connection and accept data commands sent over the socket connection. The Socket Server will parse the commands, determine what action is being requested, take the appropriate action, and finally, echo back the command as a means to confirm that the command was received and carried out.

Note: *If you open the project that is supplied with the EDK, due to possible directory differences, you will have to remove and re-add the SJJ_HardwareProvider and SJJ_PWM_Driver resources.*

10.2.3 Create the Socket Server Application

1. Open Visual Studio or Visual C# Express Edition.
2. From the menu, select **File->New->Project…**
3. The New Project dialog appears; in **Project types:** expand **Other Languages** and under **Visual C#** in the **Templates:** section under **My Templates**, you should see the SJJ_MF Console Application.
4. Click on the **SJJ_MF Console Application**.
5. Make sure **Create directory for solution** is checked, and under name enter **MF_SocketServer**.
6. Click on **OK** to create the new project.



SJJ .NET Micro Framework Application Template

Name: MF_SocketServer

Location: C:\Documents and Settings\ ... \My Documents\Visual Studio 2005\Projects\EDKplus

Solution: Create new Solution ☑ Create directory for solution

Solution Name: MF_SocketServer

Fig 10.3 Creating Socket Server Project Using SJJ Visual Studio Template

7. When you open Program.cs you will see the basic references included with the template as with earlier examples. For network and socket services, we will have to add some additional references.
8. From the menu, select **Project->Add Reference…**; in the .NET tab, click on **Microsoft.SPOT.Net**, and then click **OK**.
9. From the menu, select **Project->Add Reference…**; in the .NET tab, click on **System**, and then click **OK**.
10. Again from the menu, select **Project->Add Reference…**; select the **Browse** tab and locate and select **SJJ_HardwareProvider.dll**; click **OK**.
11. Since we will be controlling the PWM's, add the PWM managed code driver. From the menu, select **Project->Add Reference…**; select the **Browse** tab and locate and select **SJJ_PWM_Driver.dll**; click **OK**.
12. Finally, from the menu, select **Project->Add Reference…**; in the .NET tab, click on **Microsoft.SPOT.Hardware.SerialPort**; click **OK**.
13. Open Program.cs and add the following *using* directives to provide direct access to the required API's namespaces:

```
using Microsoft.SPOT.Hardware.SJJ;
using Microsoft.SPOT.Hardware.SJJ.SJJ_PWM_Driver;
using Microsoft.SPOT.Net;
using Microsoft.SPOT.Net.NetworkInformation;
using System.Net;
using System.Net.Sockets;
using Socket = System.Net.Sockets.Socket;
using System.Text;
using System.IO.Ports;
```

14. Save the project

10.2.4 Adding the Socket Server Code

Now that we have the basic application shell set up, we need to add the code to sign-on and output the IP address and socket port number, listen for connections, establish a socket connection, and receive and send data.

First, let's establish a string constant for the version number and include that in the assembly information, so that the application version can be examined from the file properties.

1. Open Program.cs and in the Program class add the following constant definition:

```
public const string c_VersionNumber = "4.0.1.3";
```

2. Open AssemblyInfo.cs and modify the Assembly Version and File version directives as follows:

```
[assembly: AssemblyVersion(SJJ_MF_Console_Application.Program.c_VersionNumber)]
[assembly: AssemblyFileVersion(SJJ_MF_Console_Application.Program.c_VersionNumber)]
```

3. Save AssemblyInfo.cs
4. In Program.cs, modify the Debug.Print in Main as follows:

```
// Sign on
Debug.Print("MF Socket Server V" + c_VersionNumber);

App myApp = new App();

// rerun server if it exits
while (true)
{
    myApp.Run();

    Debug.Print("Restarting Server");
}
```

The remainder of the sign-on and functional server code will be provided in the App class, so let's add the class members and constants that will be required.

5. In Program.cs add the following members and constants to the App class:

```
const Int32 iPort = 12000;
const int c_WaitInfinite = -1;
const int c_FiveSeconds = 5000000;
const int iReceiveBufferSize = 256;
const uint c_MinPWM1 = 225; //Hz
const uint c_uiConvertError = 999999999;
const uint c_uiMaxPWMFreq = 100000; //100 KHz
const int c_iInitDutyCycle = 50; // Initialize duty cycles to 50%
bool bRunServer = true;
int iBytesReceived = 0;
int iBytesSent = 0;
SJJ_PWM_Driver.PWM_Status eStatusPWM;

// Message constants
const string sMsgLEDOn = "LED ON";
const string sMsgLEDOff = "LED OFF";
const string sMsgHdrPWM1 = "PWM1";
const string sMsgHdrPWM2 = "PWM2";
const string sMsgHdrPWM3 = "PWM3";
const string sMsgClose = "Close";

// Control objects
// PWM driver object
SJJ_PWM_Driver myPWMDriver;
bool bPWMState;

// green LED object
OutputPort myGreenLED;
```

```
// Virtual serial port handle for PWM access
SerialPort myPwmHandle;

Socket server;
```

We are going to be controlling the green LED and the 3 PWMs, so let's do the initialization of the hardware objects in the App class constructor, since it only needs to be done once at the very beginning.

6. In Program.cs in the App class add the following constructor:

```
public App()
{
    // Create green LED GPIO and PWM objects for control accrss
    myGreenLED = new OutputPort(Pins.GREEN_LED, false);

    // Initialize PWM's and start with them off
    InitPWM(false);

    // Create a socket, bind it to the server's port and listen for client
connections
    server = new Socket(AddressFamily.InterNetwork, SocketType.Stream,
ProtocolType.Tcp);
    IPEndPoint localEndPoint = new IPEndPoint(IPAddress.Any, iPort);

    server.Bind(localEndPoint);
    server.Listen(Int32.MaxValue);
}
```

7. Then add the following PWM initialization method to the App class (*this method is called in the constructor and allows you to initialize the PWMs and turn them on or off*):

```
public void InitPWM(bool bPWMOnOff)
{
    // Create PWM object
    myPWMDriver = new SJJ_PWM_Driver();

    // Open the PWM device
    myPwmHandle = myPWMDriver.PWM_Open();

    // Turn PWM1 off
    eStatusPWM = myPWMDriver.PWM_Off(myPwmHandle);
    if (eStatusPWM != SJJ_PWM_Driver.PWM_Status.eStatusWriteSuccess)
    {
        Debug.Print("PWM_Off failed: " + eStatusPWM.ToString());
    }

    // Set PWM1 frequency to maximum for this application
    eStatusPWM = myPWMDriver.PWM_SetFreq(c_uiMaxPWMFreq, myPwmHandle);
    if (eStatusPWM != SJJ_PWM_Driver.PWM_Status.eStatusWriteSuccess)
    {
        Debug.Print("PWM_SetFreq failed: " + eStatusPWM.ToString());
    }

    // Set PWM1 duty cycle to 50%
    eStatusPWM = myPWMDriver.PWM_SetDutyCycle(SJJ_PWM_Driver.PWM_Channel.ePWM1,
c_iInitDutyCycle, myPwmHandle);
    if (eStatusPWM != SJJ_PWM_Driver.PWM_Status.eStatusWriteSuccess)
    {
        Debug.Print("PWM_SetDutyCycle ePWM1 failed: " + eStatusPWM.ToString());
    }

    // Set CPLD PWM2 duty cycle to 50%
    eStatusPWM =
myPWMDriver.PWM_SetDutyCycle(SJJ_PWM_Driver.PWM_Channel.ePWM2_CPLD,
c_iInitDutyCycle, myPwmHandle);
    if (eStatusPWM != SJJ_PWM_Driver.PWM_Status.eStatusWriteSuccess)
    {
        Debug.Print("PWM_SetDutyCycle ePWM2_CPLD failed: " +
eStatusPWM.ToString());
    }

    // Set CPLD PWM3 duty cycle to 50%
    eStatusPWM =
myPWMDriver.PWM_SetDutyCycle(SJJ_PWM_Driver.PWM_Channel.ePWM3_CPLD,
c_iInitDutyCycle, myPwmHandle);
    if (eStatusPWM != SJJ_PWM_Driver.PWM_Status.eStatusWriteSuccess)
    {
        Debug.Print("PWM_SetDutyCycle ePWM3_CPLD failed: " +
eStatusPWM.ToString());
    }

    // Turn on/off PWM
    if (bPWMOnOff)
    {
        eStatusPWM = myPWMDriver.PWM_On(myPwmHandle);
        if (eStatusPWM != SJJ_PWM_Driver.PWM_Status.eStatusWriteSuccess)
        {
            Debug.Print("PWM_On failed: " + eStatusPWM.ToString());
        }
    }
    else
    {
        eStatusPWM = myPWMDriver.PWM_Off(myPwmHandle);
        if (eStatusPWM != SJJ_PWM_Driver.PWM_Status.eStatusWriteSuccess)
        {
            Debug.Print("PWM_On failed: " + eStatusPWM.ToString());
        }
    }
}
```

Now, let's complete the sign-on, we have to output the socket port number, determine what IP address we have been assigned, and output the IP address.

8. In the Run method add the following sign-on code:

```
// Report IP Address
NetworkInterface[] mynetwork = NetworkInterface.GetAllNetworkInterfaces();

foreach (NetworkInterface mynetworkInterface in mynetwork)
{
    Debug.Print("Local IP: " + mynetworkInterface.IPAddress);
    Debug.Print("Local MAC: " +
ConvertByteToHexString(mynetworkInterface.PhysicalAddress[0]) + " " +
ConvertByteToHexString(mynetworkInterface.PhysicalAddress[1]) + " " +
ConvertByteToHexString(mynetworkInterface.PhysicalAddress[2]) + " " +
ConvertByteToHexString(mynetworkInterface.PhysicalAddress[3]) + " " +
ConvertByteToHexString(mynetworkInterface.PhysicalAddress[4]) + " " +
ConvertByteToHexString(mynetworkInterface.PhysicalAddress[5]));
}
Debug.Print("Port: " + iPort.ToString());
```

Now, let's add the meat, the functional code for the server application. Because there are a number of reasons that the socket connection could fail, we want the server code to fail gracefully if an error occurs and be able to restart and try, again. Socket failures will throw exceptions, so we will use the try-catch-finally exception handling convention of C# and .NET.

9. In the Run method, add the try block that polls for connection (*if no error occurs, it will wait forever*):

```
try
{
    // Infinite wait for a connection before exiting and failing
    bRunServer = server.Poll(c_WaitInfinite, SelectMode.SelectRead);

    if (bRunServer)
    {
            Debug.Print("Connection received");
        }
    }
```

10. In the Run method, follow the try block with the catch block to handle any exceptions that might occur due to link failure:

```
catch (Exception e)
{
    Debug.Print("Socket Exception: " + e.ToString());

    bRunServer = false;
}
```

11. Now for the meat. In the Run method following the try-catch blocks add the finally block that contains the socket message reads and writes and all the command processing:

```
finally
{
    // Wait for a client to connect
    Socket clientSocket = server.Accept();

    // 'using' ensures that the socket gets closed
    using (clientSocket)
    {

        while (bRunServer)
        {
            {   //Begin scope of receive buffer
                byte[] bReceiveBuffer = new byte[iReceiveBufferSize];

                // Check for received data
                if (clientSocket.Poll(c_FiveSeconds, SelectMode.SelectRead))
                {
                    if (clientSocket.Available == 0)
                    {
                        // We waited 5 seconds and no bytes are ready to read,
                        iBytesReceived = 0;
                    }
                    else
                    {

                        iBytesReceived = clientSocket.Receive(bReceiveBuffer,
clientSocket.Available, SocketFlags.None);

                        Debug.Print("Bytes received = " +
iBytesReceived.ToString());

                        // check for socket error
                        if (iBytesReceived < 0)
                        {
                            bRunServer = false;
                        }
                    }
                }
                else
                {
                    iBytesReceived = 0;
                }

                // loopback received data
                if (iBytesReceived > 0)
                {
                    // Parse the command received
                    // Convert byte message to string for parsing
                    string sMsgReceived = new
string(UTF8Encoding.UTF8.GetChars(bReceiveBuffer));

                    Debug.Print("Message received: " + sMsgReceived);
```

```
                            // Find substrings
                            string[] sMsgSubs = sMsgReceived.Split(null);

                            if (string.Equals(sMsgReceived, sMsgLEDOn))
                            {
                                // Turn on green LED
                                myGreenLED.Write(true);
                            }
                            else if (string.Equals(sMsgReceived, sMsgLEDOff))
                            {
                                // Turn off green LED
                                myGreenLED.Write(false);
                            }
                            else if (string.Equals(sMsgSubs[0], sMsgHdrPWM1))
                            {
                                // PWM1 command
                                uint uiSetFreq = ConvertToUint(sMsgSubs[1]);

                                if (uiSetFreq != c_uiConvertError)
                                {
                                    // check for freq == 0 and turn it off
                                    if (uiSetFreq == 0)
                                    {
                                        // Turn off PWM1
                                        eStatusPWM = myPWMDriver.PWM_Off(myPwmHandle);
                                        if (eStatusPWM !=
SJJ_PWM_Driver.PWM_Status.eStatusWriteSuccess)
                                        {
                                            Debug.Print("PWM_Off failed: " +
eStatusPWM.ToString());
                                        }
                                        else
                                        {
                                            bPWMState = false;
                                        }
                                    }
                                    else if (0 < uiSetFreq && uiSetFreq <= c_MinPWM1)
                                    {
                                        // check for 0 < freq <= c_MinPWM1 and set freq ==
c_MinPWM1 & turn on
                                        eStatusPWM = myPWMDriver.PWM_SetFreq(c_MinPWM1,
myPwmHandle);
                                        if (eStatusPWM !=
SJJ_PWM_Driver.PWM_Status.eStatusWriteSuccess)
                                        {
                                            Debug.Print("PWM_SetFreq failed: " +
eStatusPWM.ToString());
                                        }
                                        else if (!bPWMState)
                                        {
                                            // Turn on PWM
                                            eStatusPWM = myPWMDriver.PWM_On(myPwmHandle);
                                            if (eStatusPWM !=
SJJ_PWM_Driver.PWM_Status.eStatusWriteSuccess)
                                            {
                                                Debug.Print("PWM_On failed: " +
eStatusPWM.ToString());
                                            }
                                            else
                                            {
                                                bPWMState = true;
                                            }
                                        }
                                    }
                                    else
                                    {
                                        // else set freq == command freq and turn on
                                        eStatusPWM = myPWMDriver.PWM_SetFreq(uiSetFreq,
myPwmHandle);
                                        if (eStatusPWM !=
SJJ_PWM_Driver.PWM_Status.eStatusWriteSuccess)
                                        {
```

```
                                    Debug.Print("PWM_SetFreq failed: " +
eStatusPWM.ToString());
                                }
                                else if (!bPWMState)
                                {
                                    // Turn on PWM
                                    eStatusPWM = myPWMDriver.PWM_On(myPwmHandle);
                                    if (eStatusPWM !=
SJJ_PWM_Driver.PWM_Status.eStatusWriteSuccess)
                                    {
                                        Debug.Print("PWM_On failed: " +
eStatusPWM.ToString());
                                    }
                                    else
                                    {
                                        bPWMState = true;
                                    }
                                }
                            }

                            // Reset duty cycle to 50%
                            eStatusPWM =
myPWMDriver.PWM_SetDutyCycle(SJJ_PWM_Driver.PWM_Channel.ePWM1, 50, myPwmHandle);
                            if (eStatusPWM !=
SJJ_PWM_Driver.PWM_Status.eStatusWriteSuccess)
                            {
                                Debug.Print("PWM_SetDutyCycle ePWM1 failed: " +
eStatusPWM.ToString());
                            }
                        }
                        else
                        {
                            Debug.Print("Frequency string conversion error");
                        }
                    }
                    else if (string.Equals(sMsgSubs[0], sMsgHdrPWM2))
                    {
                        // PWM2 command
                        uint uiSetDutyCycle = ConvertToUint(sMsgSubs[1]);

                        if (uiSetDutyCycle != c_uiConvertError)
                        {
                            eStatusPWM =
myPWMDriver.PWM_SetDutyCycle(SJJ_PWM_Driver.PWM_Channel.ePWM2_CPLD,
(int)uiSetDutyCycle, myPwmHandle);
                            if (eStatusPWM !=
SJJ_PWM_Driver.PWM_Status.eStatusWriteSuccess)
                            {
                                Debug.Print("PWM_SetDutyCycle ePWM2_CPLD failed: "
+ eStatusPWM.ToString());
                            }
                        }
                        else
                        {
                            Debug.Print("Duty Cycle string conversion error");
                        }
                    }
                    else if (string.Equals(sMsgSubs[0], sMsgHdrPWM3))
                    {
                        // PWMe command
                        uint uiSetDutyCycle = ConvertToUint(sMsgSubs[1]);

                        if (uiSetDutyCycle != c_uiConvertError)
                        {
                            eStatusPWM =
myPWMDriver.PWM_SetDutyCycle(SJJ_PWM_Driver.PWM_Channel.ePWM3_CPLD,
(int)uiSetDutyCycle, myPwmHandle);
                            if (eStatusPWM !=
SJJ_PWM_Driver.PWM_Status.eStatusWriteSuccess)
                            {
                                Debug.Print("PWM_SetDutyCycle ePWM3_CPLD failed: "
+ eStatusPWM.ToString());
                            }
```

```
                                }
                                else
                                {
                                    Debug.Print("Duty Cycle string conversion error");
                                }
                            }
                            else if (string.Equals(sMsgReceived, sMsgClose))
                            {
                                // Socket close
                                bRunServer = false;
                            }
                            else
                            {
                                // Unknown command
                                Debug.Print("Unknown command: " + sMsgReceived);
                            }

                            // Echo back command for confirmation
                            if (iBytesReceived > 0)
                            {
                                iBytesSent = clientSocket.Send(bReceiveBuffer,
        iBytesReceived, SocketFlags.None);

                                //check for socket error
                                if (iBytesSent < 0)
                                {
                                    bRunServer = false;
                                }

                            }
                        }
                    }   //End scope of receive buffer
                }

            }

            Debug.Print("Server Exiting");
        }

        public uint ConvertToUint(string sDecimalString)
        {
            uint uiReturnVal = c_uiConvertError;
            uint uiConvertVal = 0;

            // Convert string to byte array
            byte[] byDecimalArray = Encoding.UTF8.GetBytes(sDecimalString);

            // Note byte ordering for decimal strings converted to byte arrays.
            uint uiPwrTen = 1;
            int idx;
            for (idx = (byDecimalArray.Length - 1); idx >= 0; idx--)
            {
                if ((byDecimalArray[idx] >= 0x30) && (byDecimalArray[idx] <= 0x39))
                {
                    uiConvertVal += (uint) (byDecimalArray[idx] & 0x0F) * uiPwrTen;
                    uiPwrTen *= 10;
                }
                else
                {
                    // non-decimal character in string
                    break;
                }

            }

            //Check for error
            if (idx < 0)
            {
                //Conversion success
                uiReturnVal = uiConvertVal;
            }

            return (uiReturnVal);
        }

        public string ConvertByteToHexString(byte bByte)
        {
            byte[] bHexBytes = new byte[2];

            bHexBytes[0] = (byte) (bByte >> 4);
            bHexBytes[1] = (byte) (bByte & 0x0f);

            for (int i = 0; i < 2; i++)
            {
                if (bHexBytes[i] < 0xa)
                {
                    bHexBytes[i] |= 0x30;
                }
                else
                {
                    bHexBytes[i] = (byte) ((bHexBytes[i] - 9) | 0x40);
                }
            }
            string sReturnString = new string(UTF8Encoding.UTF8.GetChars(bHexBytes));

            return (sReturnString);
        }

        public void InitPWM(bool bPWMOnOff)
        {
            // Create PWM object
            myPWMDriver = new SJJ_PWM_Driver();

            // Open the PWM device
            myPwmHandle = myPWMDriver.PWM_Open();

            // Turn PWM1 off
            eStatusPWM = myPWMDriver.PWM_Off(myPwmHandle);
            if (eStatusPWM != SJJ_PWM_Driver.PWM_Status.eStatusWriteSuccess)
            {
                Debug.Print("PWM_Off failed: " + eStatusPWM.ToString());
            }

            // Set PWM1 frequency to maximum for this application
            eStatusPWM = myPWMDriver.PWM_SetFreq(c_uiMaxPWMFreq, myPwmHandle);
            if (eStatusPWM != SJJ_PWM_Driver.PWM_Status.eStatusWriteSuccess)
            {
                Debug.Print("PWM_SetFreq failed: " + eStatusPWM.ToString());
            }

            // Set PWM1 duty cycle to 50%
            eStatusPWM = myPWMDriver.PWM_SetDutyCycle(SJJ_PWM_Driver.PWM_Channel.ePWM1,
        c_iInitDutyCycle, myPwmHandle);
            if (eStatusPWM != SJJ_PWM_Driver.PWM_Status.eStatusWriteSuccess)
            {
                Debug.Print("PWM_SetDutyCycle ePWM1 failed: " + eStatusPWM.ToString());
            }

            // Set CPLD PWM2 duty cycle to 50%
            eStatusPWM =
        myPWMDriver.PWM_SetDutyCycle(SJJ_PWM_Driver.PWM_Channel.ePWM2_CPLD,
        c_iInitDutyCycle, myPwmHandle);
            if (eStatusPWM != SJJ_PWM_Driver.PWM_Status.eStatusWriteSuccess)
            {
                Debug.Print("PWM_SetDutyCycle ePWM2_CPLD failed: " +
        eStatusPWM.ToString());
            }

            // Set CPLD PWM3 duty cycle to 50%
            eStatusPWM =
        myPWMDriver.PWM_SetDutyCycle(SJJ_PWM_Driver.PWM_Channel.ePWM3_CPLD,
        c_iInitDutyCycle, myPwmHandle);
            if (eStatusPWM != SJJ_PWM_Driver.PWM_Status.eStatusWriteSuccess)
            {
                Debug.Print("PWM_SetDutyCycle ePWM3_CPLD failed: " +
        eStatusPWM.ToString());
```

```
            }

            // Turn on/off PWM
            if (bPWMOnOff)
            {
                eStatusPWM = myPWMDriver.PWM_On(myPwmHandle);
                if (eStatusPWM != SJJ_PWM_Driver.PWM_Status.eStatusWriteSuccess)
                {
                    Debug.Print("PWM_On failed: " + eStatusPWM.ToString());
                }
            }
            else
            {
                eStatusPWM = myPWMDriver.PWM_Off(myPwmHandle);
                if (eStatusPWM != SJJ_PWM_Driver.PWM_Status.eStatusWriteSuccess)
                {
                    Debug.Print("PWM_On failed: " + eStatusPWM.ToString());
                }
            }
        }
```

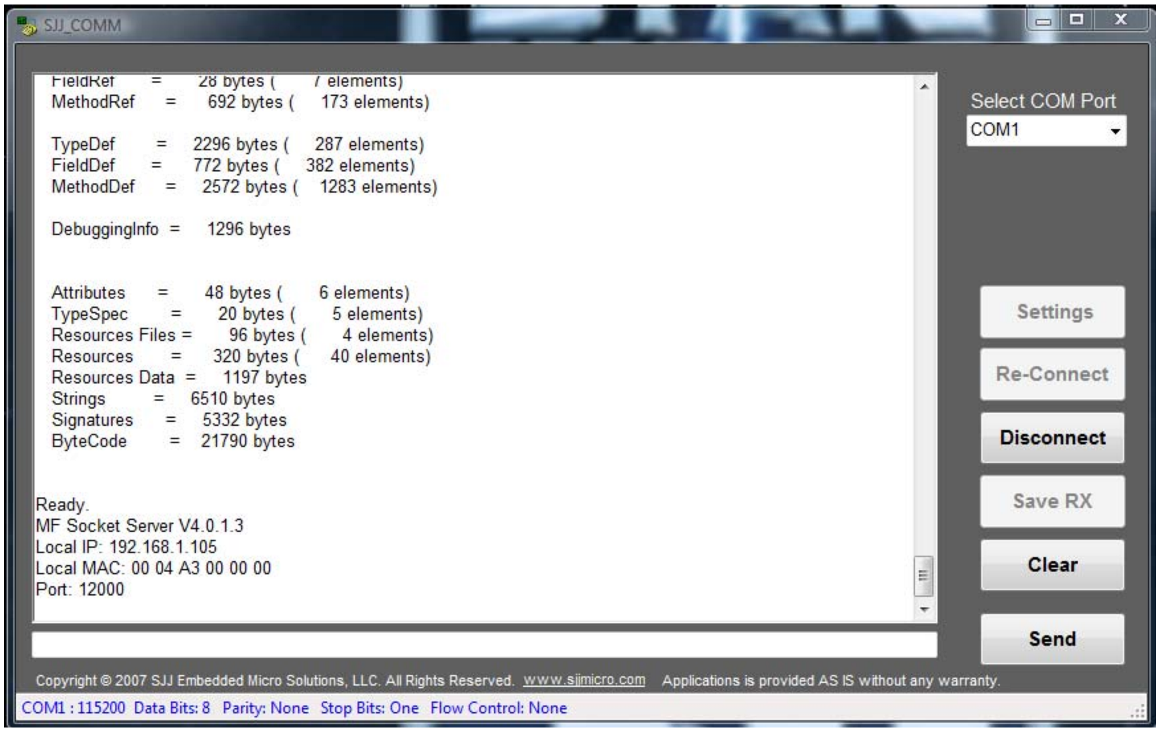
10.2.5 Configure Project Properties, Build, and Deploy the application

1. Make sure the Null modem cable is connected to the debug serial port of the iPac-9302 and the development computer.
2. With the **MF_SocketServer** project selected in the Solution Explorer, from the menu, select **Project** and then select **MF_SocketServer Properties…** from the submenu.
3. The project properties page appears, click on the **Micro Framework** on the left tab.
4. Under Deployment, select **Serial** for the Transport and **Com#** as the Device. Where # is the COM port number of the development computer.
5. Save the project
6. With the **Solution 'MF_SocketServer'** selected in the Solution Explorer, from the menu, select **Project** and then select **Properties** from the submenu.
7. The Solution Property Pages appears. Expand **Configuration Properties** on the left and select **Configuration**. Make sure that the **Build** and **Deploy** checkboxes are **checked** in the **Project contexts**. Click **OK**.
8. Save the solution.
9. From the menu, select **Build**, and then select **Build Solution** from the drop down menu. There should be no build errors.
10. Make sure the iPac-9302 is powered on. You need to wait at least 20 seconds before deploying an application. This gives TinyBooter a chance to turn control over to the CLR. You can also monitor the boot-up of the iPac-9302 with MFDeploy and use the clearing of the BootLoader Flag to shorten the boot time, but **remember to disconnect MFDeploy before you attempt to deploy an application from Visual Studio.**
11. From the menu, select **Build**, and then select **Deploy Solution**.

The output window should show a successful download.

10.2.6 Running MF_SocketServer

Make sure that the debug COM port of the iPac-9302 is connected to the development workstation. Launch a serial terminal program (*you can use SJJ COM*) and connect to the COM port that the iPac-9302 is connected to. Configure the COM parameters to be 115200 baud, 8 data bits, no parity, and 1 stop bit. Repower the iPac-9302 and after the MF_SocketServer boots up and acquires an IP address you should see output on the terminal program similar to Fig 10.4



```
 FieldRef    =    28 bytes (      7 elements)
 MethodRef   =    692 bytes (   173 elements)

 TypeDef     =   2296 bytes (   287 elements)
 FieldDef    =   772 bytes (   382 elements)
 MethodDef   =   2572 bytes (   1283 elements)

 DebuggingInfo =   1296 bytes


 Attributes    =    48 bytes (      6 elements)
 TypeSpec      =     20 bytes (      5 elements)
 Resources Files =     96 bytes (      4 elements)
 Resources     =    320 bytes (    40 elements)
 Resources Data  =   1197 bytes
 Strings       =   6510 bytes
 Signatures    =    5332 bytes
 ByteCode      =   21790 bytes


Ready.
MF Socket Server V4.0.1.3
Local IP: 192.168.1.105
Local MAC: 00 04 A3 00 00 00
Port: 12000
```

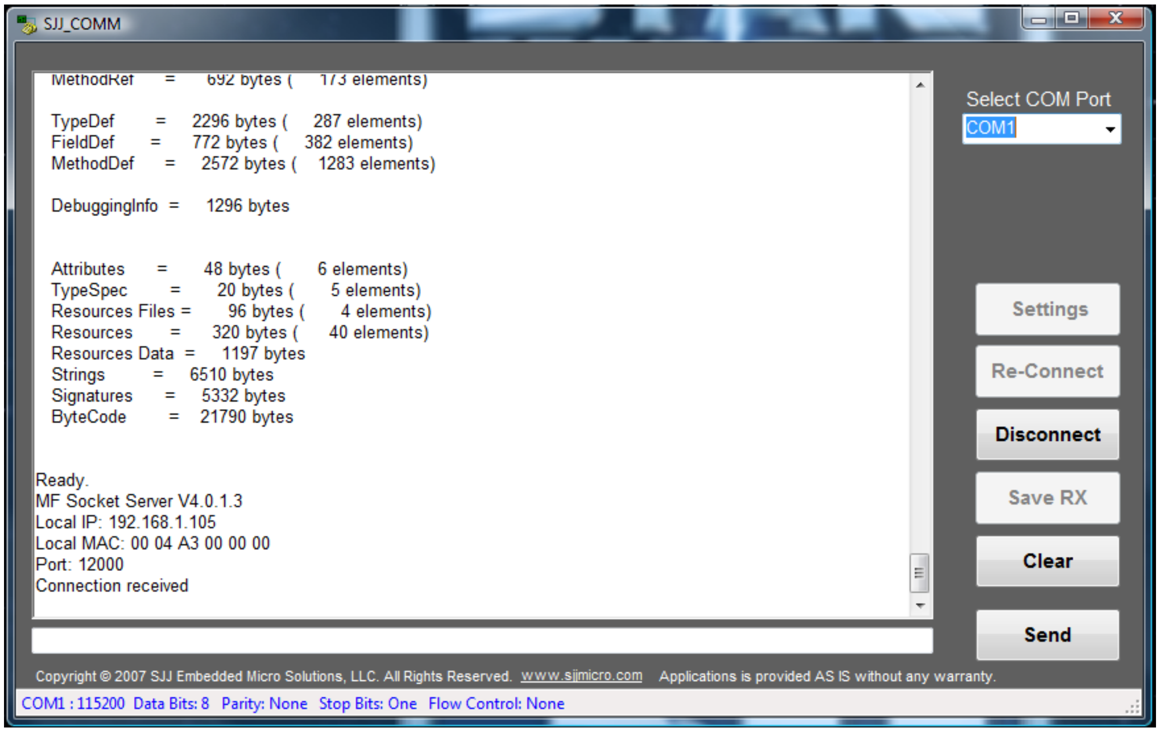
Copyright © 2007 SJJ Embedded Micro Solutions, LLC. All Rights Reserved. www.sjjmicro.com Applications is provided AS IS without any warranty.

COM1 : 115200 Data Bits: 8 Parity: None Stop Bits: One Flow Control: None

Fig 10.4 MF_SocketServer Boot-Up Debug Output

Note the 4 lines after “Ready”. The Socket Server signs on with its name and version, it identifies the IP address and MAC that it has acquired, and finally it shows the port number it is listening on. Launch the Desktop Client and enter the IP address into the IP Address TextBox. Confirm that the Socket number shown in the Desktop Client Socket TextBox is the same as what was output by the Socket Server.

When the IP Address and Socket numbers are set to match between the Desktop Client and Socket Server, click the **Connect** button on the Desktop Client. When the connection is processed the Desktop Client will report it in the status TextBox and the Socket Server will also report it via the debug port, see Fig 10.5 and Fig 10.6.

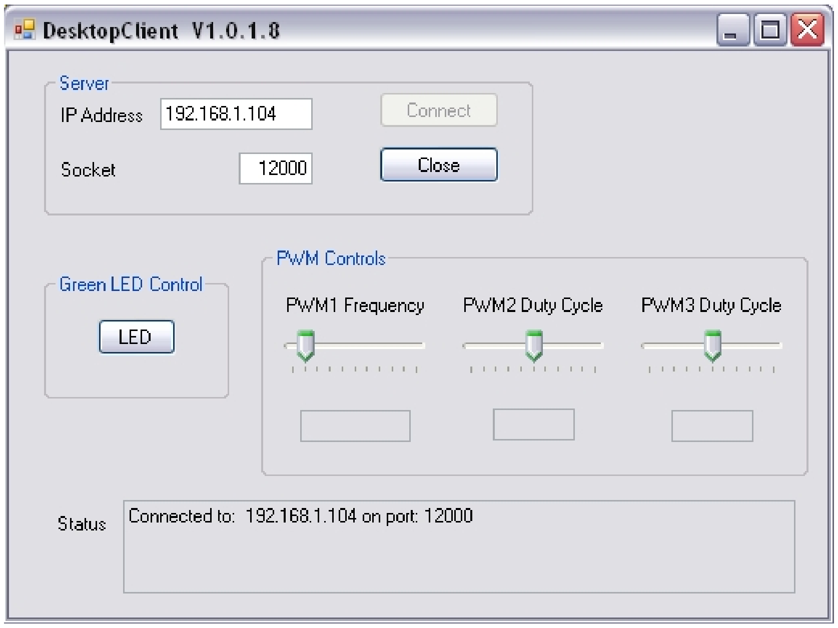


```
   MethodRef     =     692 bytes (    173 elements)

   TypeDef       =    2296 bytes (    287 elements)
   FieldDef      =     772 bytes (    382 elements)
   MethodDef     =    2572 bytes (   1283 elements)

   DebuggingInfo =    1296 bytes


   Attributes    =      48 bytes (      6 elements)
   TypeSpec      =      20 bytes (      5 elements)
   Resources Files =    96 bytes (      4 elements)
   Resources     =     320 bytes (     40 elements)
   Resources Data =   1197 bytes
   Strings       =    6510 bytes
   Signatures    =    5332 bytes
   ByteCode      =   21790 bytes


Ready.
MF Socket Server V4.0.1.3
Local IP: 192.168.1.105
Local MAC: 00 04 A3 00 00 00
Port: 12000
Connection received
```

Copyright © 2007 SJJ Embedded Micro Solutions, LLC. All Rights Reserved. www.sjjmicro.com Applications is provided AS IS without any warranty.

COM1 : 115200 Data Bits: 8 Parity: None Stop Bits: One Flow Control: None

Fig 10.5 Socket Server Connection Received



IP Address: 192.168.1.104

Socket: 12000

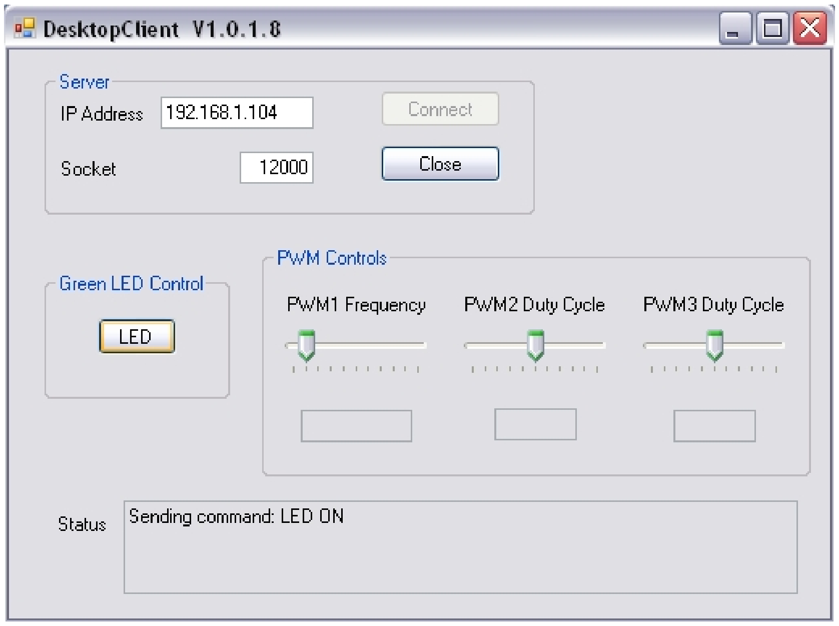
Status: Connected to: 192.168.1.104 on port: 12000

Fig 10.6 Desktop Client Connected

At this point we have initialize the board on boot-up to have the green LED off, the PWM1 frequency set to the maximum for this exercise, 100 KHz, the PWM1 duty cycle set to 50%, and the PWM1 off. PWM2 and PWM3 are initialized to a duty cycle of 50% each, but because PWM1 is off, which is the reference clock for PWM2 and PWM3 then they are off as well. After the connection is established, the Desktop Client should look like Fig 10.6:

Note: *As the Desktop Client polls for messages from the Socket Server, it will report "No bytes received from the server" each time the poll period expires and nothing is received from the server.*

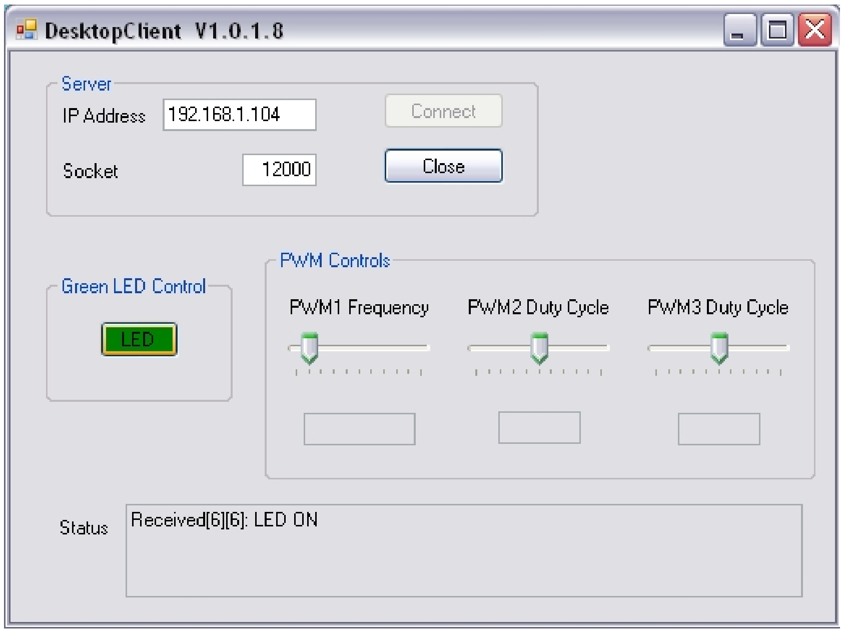
Let's start by switching the green LED. Remember we have implemented a complete client/server command scheme with command confirmation, so when we click the LED button the command will be sent out, but the state of the LED as reported by the Desktop Client will not change until it receives confirmation from the server. Therefore, click the **LED** button and you will see the command that is being sent as a Sockets message displayed in the Status Text box, see Fig 10.7:



IP Address: 192.168.1.104
Socket: 12000
Status: Sending command: LED ON

Fig 10.7 LED Button Clicked; LED ON Command Sent

When the Socket Server receives, parses, and acts on the command, it will turn the green LED on and then echo back the command message as confirmation. When the Desktop Client receives the echoed command and parses it, it will change the color of the LED button from the background gray to green and it will display the echoed message, see Fig 10.8:



Received[6][6]: LED ON

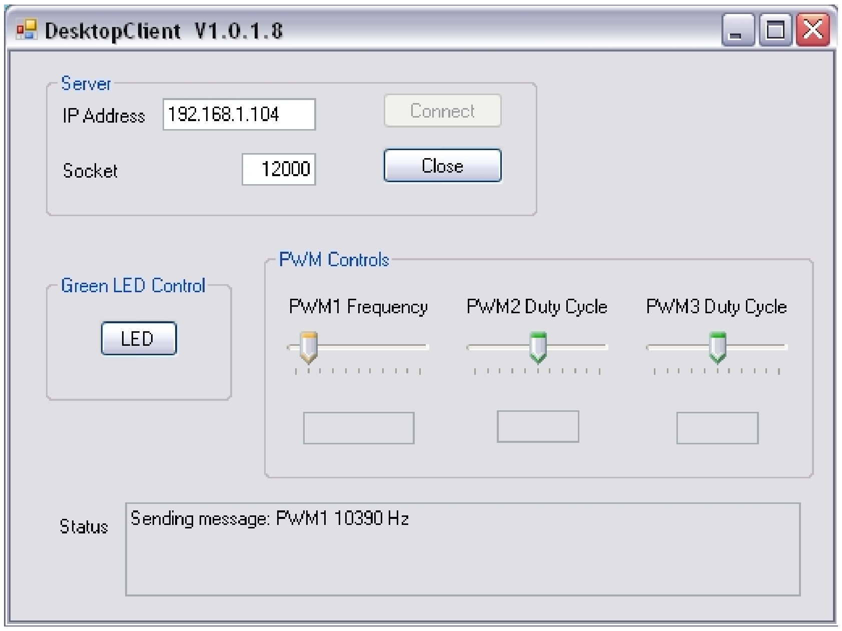
Fig 10.8 LED ON Command Confirmation Received

Note: *In the Status TextBox you will see Received[n][m]: <echoed command>. The numbers in the square brackets are for diagnostic purposes. If there were errors, it is possible that a message with non-printable characters might be received; therefore, the number 'n' is the number of bytes that were received when the message was received and the number 'm' is the number of characters in the string that was displayed after the byte array message was converted to a string class object. If the 'n' does not equal 'm' or if they are different from the number of characters you see displayed in the Status TextBox, then this would indicate a possible error.*

If you click the LED button, again, the LED OFF command will be sent, the LED will be turned off by the Socket Server, and on receipt of the echoed command, the color of the LED button would revert to gray.

Now, let's exercise the PWM TrackBar controls and observe the reaction. For the illustrations in this manual we have connected an oscilloscope to the 3 PWM outputs, so we can show the correlation between the control settings and the PWM outputs. PWM1 output will be displayed on oscilloscope channel 1 and can be sampled on pin 29 of HDR3. PWM2 output will be displayed on oscilloscope channel 2 and can be sampled on pin 15 of HDR2. Finally, PWM3 output will be displayed on oscilloscope channel 3 and can be sampled on pin 16 of HDR2.

We initialized the PWM's to be off, and since we have not gotten any PWM confirmation messages from the Socket Server, all 3 PWM TextBoxes are blank. We will start with PWM1 frequency, so we can turn the whole PWM system on. If you click on the slider of the PWM1 TrackBar, the frequency value that the control is selecting will be sent when the mouse is released (you can click, hold and move the slider back and forth and no commands will be sent until the mouse button is released). You can simply click the slider where it is without moving it, and that will send its current value. As a first test, let's do that. Click the PWM1 control and you will observe by the status message that the PWM1 command is being sent. See Fig 10.9



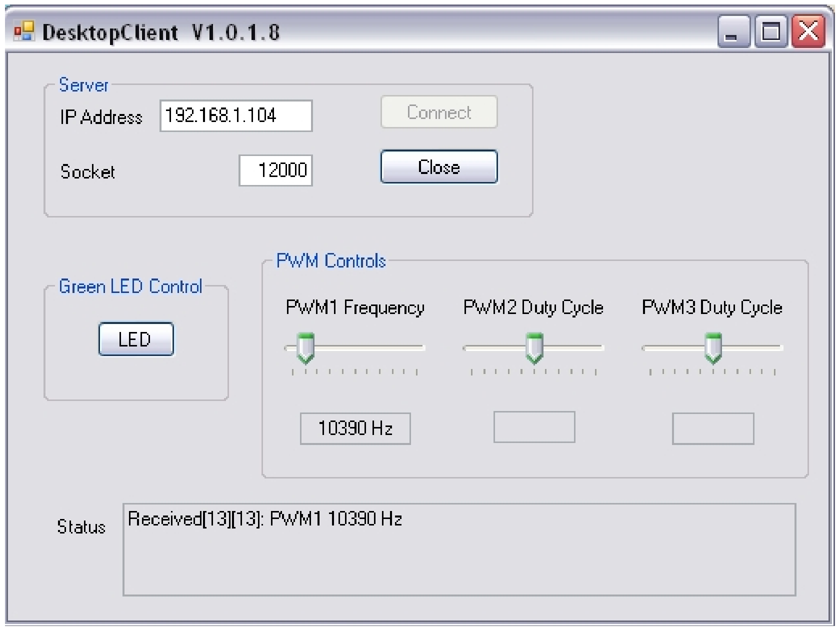
IP Address 192.168.1.104

Socket 12000

Status: Sending message: PWM1 10390 Hz

Fig 10.9 PWM1 TrackBar Slider Clicked

As with all the controls, the TextBox will not be updated until the Socket Server sets the new PWM1 frequency and echoes back the command for confirmation. When the confirmation is received, you will see the TextBox updated as in Fig 10.10.



IP Address 192.168.1.104

Socket 12000

PWM1 Frequency: 10390 Hz

Status: Received[13][13]: PWM1 10390 Hz

Fig 10.10 PWM1 Frequency Command Confirmed

The command requested a frequency of 10.390 KHz be set for PWM1. Checking the PWM outputs on the oscilloscope shows PWM1 actual frequency is measured as 10.37 KHz, well within the accuracy limit of the iPac-9302 hardware and the accuracy of the oscilloscope measurement. The Socket Server always sets the PWM1 to a duty cycle of 50%, see Fig 10.11

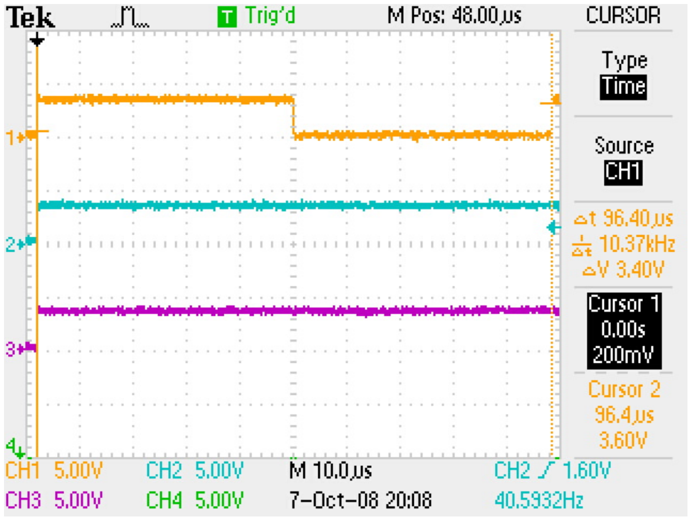


Fig 10.11 PWM1 Frequency Measurement

We know that when we turn on PWM1 we also turn on PWM2 and PWM3 and their frequencies will be the PWM1 frequency divided by 256. If we contract the time base of the oscilloscope, we can view the PWM2 & 3 outputs. Fig 10.12 Shows both PWM2 and PWM3 to be running at 40.65 Hz, which is within the accuracy range for the expected 40.51 Hz expected.

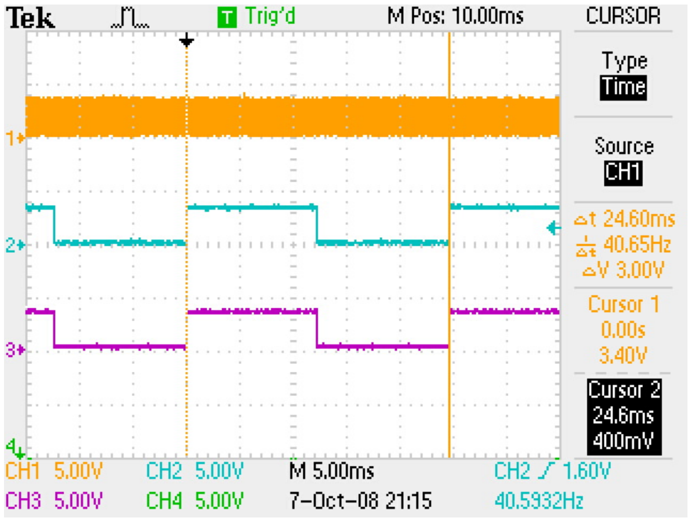
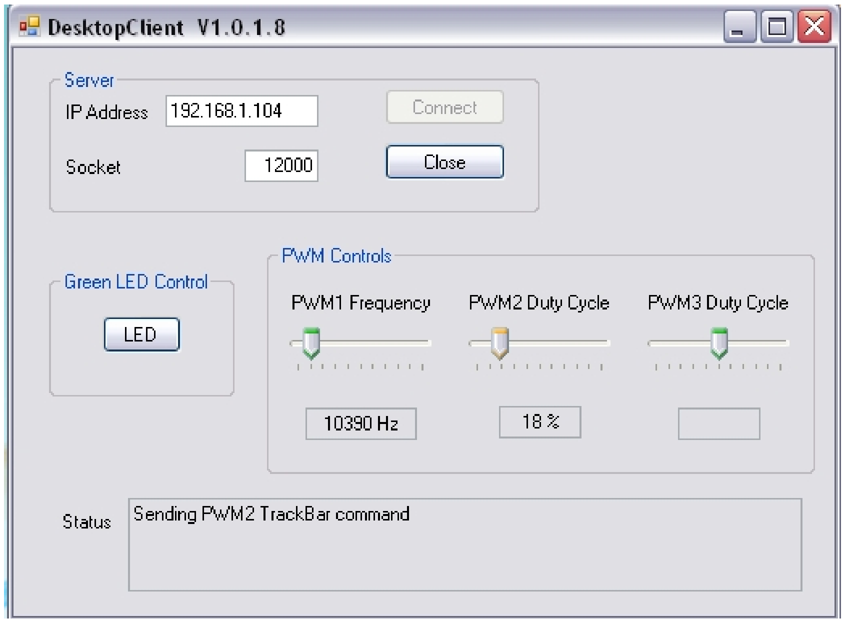


Fig 10.12 PWM2 and PWM3

Note: *When PWM1 was turned on, the Socket Server did not inform the Desktop Client of the PWM2 or PWM3 duty cycles, so those TextBoxes are still blank. This is a simple exercise, so we will leave that for an exercise for you to do if you want to experiment with enhancing the code.*

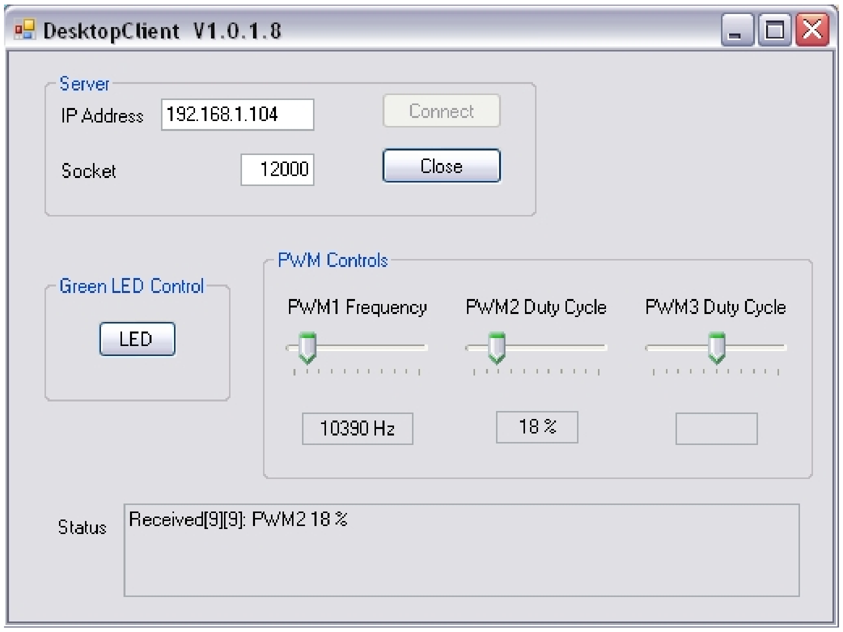
Let's now test the duty cycle controls. Click on and drag the slider for the PWM2 duty cycle and drag it to the left, reducing the duty cycle. As with the PWM1 control, first you will see the indication of the command being sent in the Status TextBox, see Fig 10.13



IP Address: 192.168.1.104
Socket: 12000
PWM1 Frequency: 10390 Hz
PWM2 Duty Cycle: 18 %
Status: Sending PWM2 TrackBar command

Fig 10.13 PWM2 Send Duty Cycle Change Command

Then when the confirmation is received, the PWM2 TextBox will be updated to show, in this case, 18% and the echoed command will be displayed in the Status TextBox, see Fig 10.14



IP Address: 192.168.1.104
Socket: 12000
PWM1 Frequency: 10390 Hz
PWM2 Duty Cycle: 18 %
Status: Received[9][9]: PWM2 18 %

Fig 10.14 PWM2 Duty Cycle Change Confirmed

Let's examine the outputs, and we will see that PWM2 has, indeed, changed and the duty cycle has been reduced. The full time between pulses is measured to be 24.6 mSec on the oscilloscope, see Fig 10.15

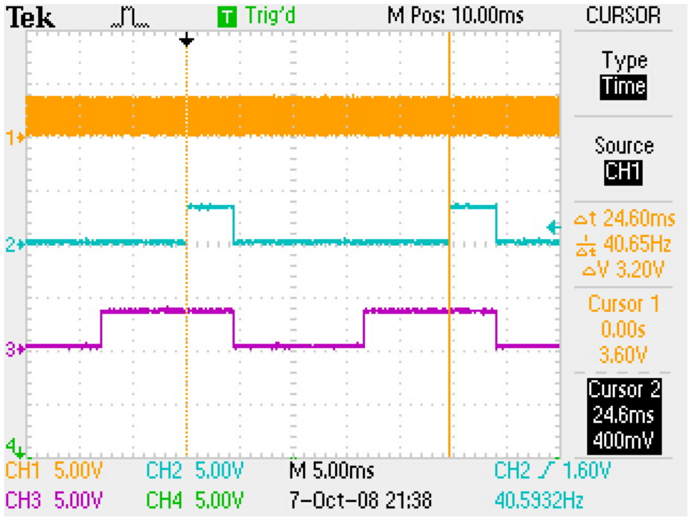


Fig 10.15 PWM2 Pulse Period

The on-time of each pulse is measured to be 4.4 mSec on the oscilloscope, see Fig 10.16

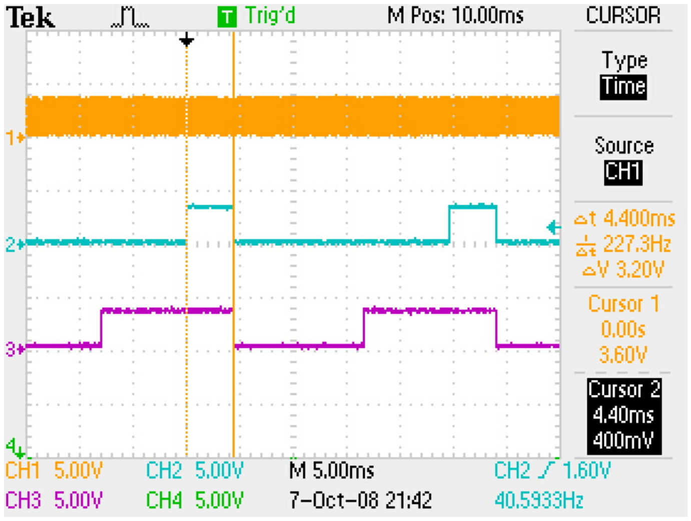
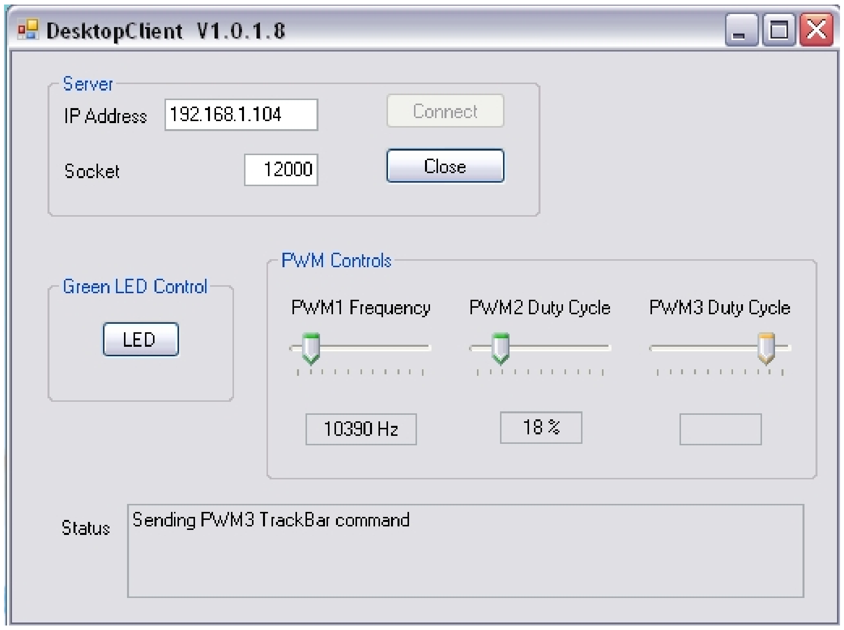


Fig 10.16 PWM2 Pulse On-Time

These measurements yield a duty cycle of 17.9%, agreeing within accuracy limits with the control setting on the Desktop Client.

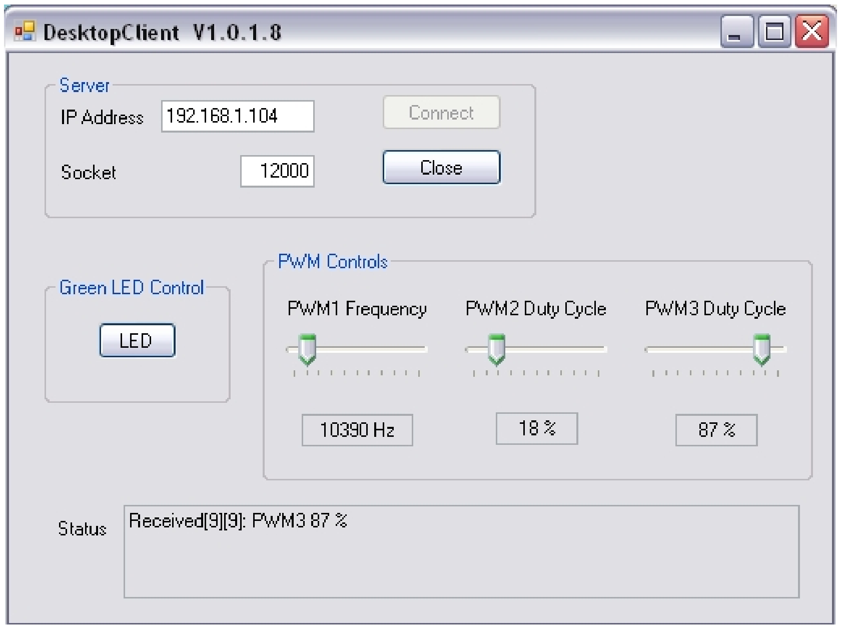
Let's do the same thing with PWM3 only increase its duty cycle. Click on and drag the slider for the PWM3 duty cycle and drag it to the right, increasing the duty cycle. As with the PWM2 control, first you will see the indication of the command being sent in the Status TextBox, see Fig 10.17



IP Address: 192.168.1.104
Socket: 12000
PWM1 Frequency: 10390 Hz
PWM2 Duty Cycle: 18 %
PWM3 Duty Cycle:
Status: Sending PWM3 TrackBar command

Fig 10.17 PWM3 Send Duty Cycle Change Command

Then when the confirmation is received, the PWM3 TextBox will be updated to show, in this case, 87% and the echoed command will be displayed in the Status TextBox, see Fig 10.18



IP Address: 192.168.1.104
Socket: 12000
PWM1 Frequency: 10390 Hz
PWM2 Duty Cycle: 18 %
PWM3 Duty Cycle: 87 %
Status: Received[9][9]: PWM3 87 %

Fig 10.18 PWM3 Duty Cycle Change Confirmed

Let's examine the outputs, again, and we will see that PWM3 has, indeed, changed and the duty cycle has been increased. The full time between pulses is measured to be, again, 24.6 mSec on the oscilloscope, see Fig 10.19

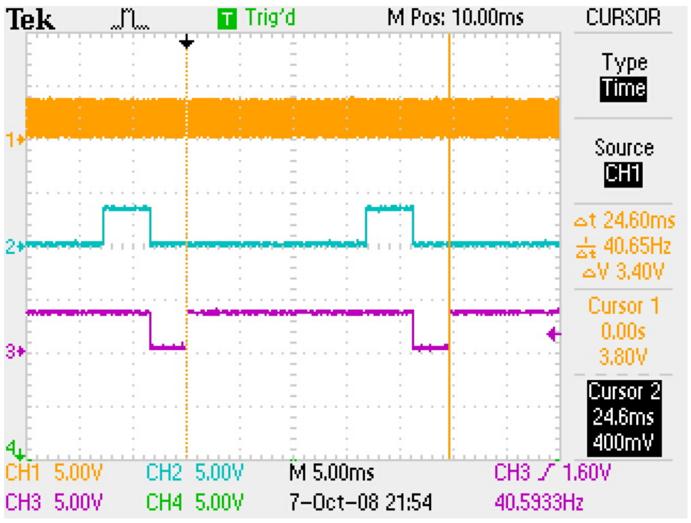


Fig 10.19 PWM3 Pulse Period

The on-time of each pulse is measured to be 21.2 mSec on the oscilloscope, see Fig 10.20

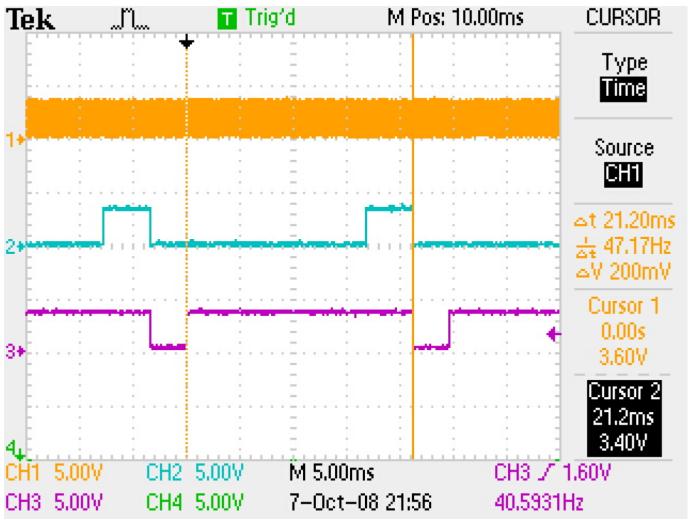


Fig 10.20 PWM3 Pulse On-Time

These measurements yield a duty cycle of 86.2%, agreeing with accuracy limits with the control setting on the Desktop Client.

10.3 SUMMARY: Ethernet, TCP/IP, and Sockets

The iPac-9302 is equipped with an Ethernet controller and the .NET Micro Framework CLR provides a TCP/IP stack and subset of Socket services that allow .NET MF devices to be interconnected over an Ethernet network. Full bidirectional messaging is supported which allows for a variety of client and server scenarios to be implemented.

11 Other Topics and Ideas

In every book or guide, we have written there always winds up to be a chapter of miscellaneous topics to discuss. With a multipurpose platform, this could be a long chapter, but we will limit ourselves to just a few ideas.

11.1 *Exercise 11.1: GPIO Block Access Library*

The iPac-9302 comes with multiple GPIO ports. The CPLD offers 2 sets of 8 GPIO outputs and 8 GPIO inputs. The EP9302 has several GPIO ports available. Let's say you have a device that has an I/O port of individual signals; for example, an ADC0804 A/D converter chip with 8 output lines. Connecting the output port to iPac-9302 GPIOs is an obvious choice. Programmatically setting up individual GPIOs would be a bit messy. Like the SPI LCD Driver, we can create a managed class driver that handles groups of GPIOs, where we can just send or receive and integer value.

In the Chapter 8 folder, there is a Block_GPIO_Driver project, which groups the CPLD GPIOs into PW_Input, PX_Output, PY_Output, and PZ_Input methods.

1. Open Visual Studio or Visual C# Express Edition.
2. Open the Block_GPIO_Driver project
3. There are 4 regions that have a block of code for each of the CPLD ports. Regions help specify a block of code that can be collapsed in the editor. This feature helps with grouping like items together and hiding code. If it is not already expanded, expand the PW_GPIO.

```
InputPort PW0 = new InputPort(Pins.GPIO_PORT_W_0, false, ResistorMode.Disabled);
InputPort PW1 = new InputPort(Pins.GPIO_PORT_W_1, false, ResistorMode.Disabled);
InputPort PW2 = new InputPort(Pins.GPIO_PORT_W_2, false, ResistorMode.Disabled);
InputPort PW3 = new InputPort(Pins.GPIO_PORT_W_3, false, ResistorMode.Disabled);
InputPort PW4 = new InputPort(Pins.GPIO_PORT_W_4, false, ResistorMode.Disabled);
InputPort PW5 = new InputPort(Pins.GPIO_PORT_W_5, false, ResistorMode.Disabled);
InputPort PW6 = new InputPort(Pins.GPIO_PORT_W_6, false, ResistorMode.Disabled);
InputPort PW7 = new InputPort(Pins.GPIO_PORT_W_7, false, ResistorMode.Disabled);

public int PW_Input()
{

    int PW = 0;

    if (PW0.Read())
    {
        PW |= 1;
    }
    if (PW1.Read())
    {
        PW |= 2;
    }
    if (PW2.Read())
    {
        PW |= 4;
    }
    if (PW3.Read())
    {
        PW |= 8;
    }
    if (PW4.Read())
    {
        PW |= 0x10;
    }
    if (PW5.Read())
    {
        PW |= 0x20;
    }
```

```
        if (PW6.Read())
        {
             PW |= 0x40;
        }
        if (PW7.Read())
        {
            PW |= 0x80;
        }

        return PW;

    }
```

The first thing is to instantiate the individual PW GPIO pins. The PW_Input method returns an integer based on the values found on each of the GPIO pins. The GPIO pins are considered to be in the order of PW0 as the least significant bit (LSB) and PW7 as the most significant bit (MSB). Each of the GPIO pins is read to see if there is a 1 or 0 on the pin. If there is a 1, then 'or' in the logic 1 into the bit location of the PW value.

4. Expand the PY_GPIO Region.

```
    OutputPort PY0 = new OutputPort(Pins.GPIO_PORT_Y_0, false);
    OutputPort PY1 = new OutputPort(Pins.GPIO_PORT_Y_1, false);
    OutputPort PY2 = new OutputPort(Pins.GPIO_PORT_Y_2, false);
    OutputPort PY3 = new OutputPort(Pins.GPIO_PORT_Y_3, false);
    OutputPort PY4 = new OutputPort(Pins.GPIO_PORT_Y_4, false);
    OutputPort PY5 = new OutputPort(Pins.GPIO_PORT_Y_5, false);
    OutputPort PY6 = new OutputPort(Pins.GPIO_PORT_Y_6, false);
    OutputPort PY7 = new OutputPort(Pins.GPIO_PORT_Y_7, false);

    public void PY_Output(int PY)
    {

         if ((PY & 1) > 0)
        {
            PY0.Write(true);
        }
        else
        {
            PY0.Write(false);
        }

        if (((PY >> 1) & 1) > 0)
        {
            PY1.Write(true);
        }
        else
        {
            PY1.Write(false);
        }

        if (((PY >> 2) & 1) > 0)
        {
            PY2.Write(true);
        }
        else
        {
            PY2.Write(false);
        }

        if (((PY >> 3) & 1) > 0)
        {
            PY3.Write(true);
        }
        else
        {
            PY3.Write(false);
        }
```

```
            if (((PY >> 4) & 1) > 0)
            {
                PY4.Write(true);
            }
            else
            {
                PY4.Write(false);
            }

            if (((PY >> 5) & 1) > 0)
            {
                PY5.Write(true);
            }
            else
            {
                PY5.Write(false);
            }

            if (((PY >> 6) & 1) > 0)
            {
                PY6.Write(true);
            }
            else
            {
                PY6.Write(false);
            }

            if (((PY >> 7) & 1) > 0)
            {
                PY7.Write(true);
            }
            else
            {
                PY7.Write(false);
            }

        }
```

Like the input pins of PW, the output pins of PY are defined. Here is the thing to watch out for: notice that we defined all the output pins to be initialized as false. The initial state of the pin may affect any devices attached to the pin. Architecture was stressed in the previous chapter, and care should be taken to understand what the state of each GPIO needs to be at startup. This is why we listed the boot state of each pin in Chapter 5 General Purpose Input/Output Pins.

The PY_Output method takes the integer value passed in and tests the individual bits through a shift and bit-and test and sets each GPIO output pin to its corresponding bit value.

Now let’s test the driver by running the application in the iPac-9302 emulator.

5. Close the project.
6. Open the Block_Driver_Test project.
7. You may have to re-reference the SJJ Hardware Provider and the Block_GPIO_Driver DLLs based on the location of these components in your system.

In the App class, a GPIO port is defined for input. A reference to the Block_GPIO_Driver is defined.

```
        InputPort EGPIO1 = new InputPort(Pins. EGPIO1_HDR3_13_CLK1_HZ, false,
        ResistorMode.Disabled);
        Block_GPIO_Driver myCPLD = new Block_GPIO_Driver();
```

The main body of the application is an infinite while-loop that checks to see if EGPIO1 is set to 1. If it is set to 1, then the GPIO lines on PW inputs are sent out to PY outputs, and the PZ inputs are sent to PX outputs.

```
while (true)
{
    if (EGPIO1.Read())
    {

        myCPLD.PY_Output(myCPLD.PW_Input());
        Debug.Print("PW Input: " + myCPLD.PW_Input().ToString());

        myCPLD.PX_Output(myCPLD.PZ_Input());
        Debug.Print("PZ Input: " + myCPLD.PZ_Input().ToString());

    }

    Thread.Sleep(5000);
}
```

8. With the project selected in the Solution Explorer, from the menu, select **Project** and then select **Block_Driver_Test Properties…** from the submenu.
9. The project properties page appears, click on the **Micro Framework** tab.
10. Under Deployment, select **Emulator** for the Transport and **iPac-9302 Emulator** as the Device.
11. Save the project
12. From the menu, select **Build**, and then select **Build Block_Driver_Test** from the drop down menu.
13. From the menu, select **Build**, and then select **Deploy Block_Driver_Test.**
14. Click on **F5**, or from the Debug menu, click **Start Debugging**.
15. Once the emulator starts, click on a few of the PW and PZ check boxes, and then click on the EGPIO1 check box. You will have to wait a few seconds, and the corresponding PY and PX outputs pins should changes, as well as, the Debug output for each of the PW and PZ values.
16. Uncheck EGPIO1, and make other changes to the PW and PZ input pins, and then click on EGPIO1 again. The outputs should change.

If we didn't have the Block_GPIO_Driver, the coding for this application would be lengthy. With the Block_GPIO_Driver, the application itself is easier to write and manage, plus the driver can be re-used for other applications and can be re-coded to support the CPU GPIO ports as well. The two things to keep in mind are the initial configuration of the GPIO pins when it comes to initial state for output pins and glitch filtering for input pins, and that the output GPIO pins are not a true parallel port and all pins do not switch states simultaneously nor are they sampled simultaneously. It is important, when designing your system, to make sure that the port that you are reading, will be stabile and unchanging during the read period. When writing a port, you want to be sure that the hardware does not sample the output until all bits have been written. You might want to use addition GPIOs as semaphores.

11.2 Some Programming Tips

Showing you what to do is important, but showing what not to do is equally important and can save you time from coding something the wrong way. Here are few tips to be aware of as you use .NET MF:

11.2.1 Application End Loop

While-loops have been used to keep the application running. You can have an application just end, but that will stop the system. The system would have to be restarted to rerun the application.

If you want to wait on interrupts, an alternative is needed. There are two. First you can use the while (true) loop to make an infinite loop:

```
while (true)
{
    // Do some stuff
}
```

The second solution is to use the thread sleep forever:

```
Thread.sleep(-1);
```

11.2.2 To Debug.Print or Not to Debug.Print; That is the Question

Debug.Print sends data and debug information out the debug serial port. Since the iPac-9302 uses the debug serial port to download new applications into flash, it is not always a good idea to add a lot of Debug.Print statements in periodic loops. When Visual Studio connects with the iPac-9302 to deploy an application, the .NET MF will stop the current application and communicate with Visual Studio to download and flash the new application; but if the application has the COM port very busy with debug output, Visual Studio cannot connect with the .NET MF and cannot stop the application, so the deploy operation will ultimately fail. For example if the GreenLED project were to be modified to the following:

```
while (true)
{
    Thread.Sleep(500);
    myGreenLED.Write(true);
    Debug.Print("Hello");

    Thread.Sleep(500);
    myGreenLED.Write(false);
    Debug.Print("World");
}
```

Since the while-loop is an infinite loop at the end of the application, the Debug.Print would probably collide with any attempts to download to the device. If you have no choice and need to have a lot of debug output, recognize that this will cause a problem with the download and deploy, so you will have to use MFDeploy to erase the application section in flash before downloading a new application.

11.2.3 Interrupt Service Routine Mistakes

We want to keep the tasks in the ISR very short. Adding long Thread.sleeps or Debug.Print would NOT be good programming practice:

Bad Example 1:

```
void EGPIO_OnInterrupt(Cpu.Pin port, bool state, TimeSpan time)
{
    Debug.Print("Interrupt hit");
    myGreenLED.Write(true);
    myPY7.Write(false); // Go low for the next interrupt

}
```

Bad Example 2:

```
        void EGPIO_OnInterrupt(Cpu.Pin port, bool state, TimeSpan time)
        {
            Thread.Sleep(200);
            myGreenLED.Write(true);
            myPY7.Write(false); // Go low for the next interrupt

        }
```

11.2.4 Clean Instantiation

When creating an instance of the hardware object, we sometimes separated the definition from the instantiation:

```
        OutputPort myGreenLED;
        public void RUN()
        {
            myGreenLED = new OutputPort(Pins.GREEN_LED, false);
            .
            .
            .
```

A cleaner solution that aids in code being more self-documenting is to do the following:

```
        OutputPort myGreenLED = new OutputPort(Pins.GREEN_LED, false);
        public void RUN()
        {
            .
            .
            .
```

11.3 Summary - Reusable Code Delivered

As we opened in Chapter 1, we said that reusable code was one of the many goals in modern programming. .NET allows you to re-use pre-defined and custom made objects in your application. If we didn't have the SPI LCD driver or the Block GPIO driver, the application code using these objects would be considerably larger and more complicated.

A C# Primer

Like any programming language, C# supports the usual operators, program flow commands, arrays, value types, and error handling. The information supplied here is a brief overview for reference. You can find more information about C# programming in many of the books that are listed in the Bibliography or online at http://msdn2.microsoft.com/en-us/library/aa287558(VS.71).aspx.

A.1 C# Operators

C# provides a large set of operators. The operators are symbols that specify which operation to perform in an expression such arithmetic, logical, shift, conditional, etc. C# has the usual arithmetic, increment/decrement, shift, and logical operators, as well as, as a variety of others shown in the following table. In addition, many operators can be overloaded by the user, thus changing their meaning when applied to a user defined type.

Operator Category	Operators	Comments
Arithmetic	+ - * / %	Basic arithmetic, but order of operation must be observed. % - gets there remainder Arithmetic overflow can result in an overflow exception.
Logical (boolean and bitwise)	& | ^ ! ~ && || true false	Boolean test logic and enumerated types
String concatenation	+	Merge strings together.
Increment, decrement	++ --	
Shift	<< >>	Shift but doesn't change original variable
Relational	== != < > <= >=	Test conditions
Assignment	= += -= *= /= %= &= |= ^= <<= >>=	
Member access	.	Namespace
Indexing	[]	For Arrays, multidimensional arrays not supported.
Cast	()	Conversion from one value type to another
Conditional	? :	If (condition) then A else B X = Y = 0 ? 12 : 6 If Y is 0, then X= 12 If Y is not 0 then X = 6
Delegate concatenation and removal	+ -	
Object creation	new	
Type information	as is sizeof typeof	
Overflow exception control	checked unchecked	
Indirection and Address	* -> [] &	

Table A 1 - C# Operators

A.2 Value Types

C# supports the following value types:

Value Type	Description
Bool	Boolean value – either true or false
Byte	8-bit unsigned integer ranging from 0 to 255
Char	Unicode 16-bit character
Decimal	128-bit data type, range $\pm 1.0 \times 10^{-28}$ to $\pm 7.9 \times 10^{28}$
Double	64-bit floating-point values, ranging $\pm 5.0 \times 10^{-324}$ to $\pm 1.7 \times 10^{308}$
Float	32-bit floating-point values, ranging $\pm 1.5 \times 10^{-45}$ to $\pm 3.4 \times 10^{38}$
Int	Signed 32-bit integer, range -2,147,483,648 to 2,147,483,647
Long	Signed 64-bit integer, range –9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
Sbyte	Signed 8-bit integer, range -128 to 127
Short	Signed 16-bit integer, range -32,768 to 32,767
Struct	Encapsulate a group of variables
Unit	Unsigned 32-bit integer, range 0 to 4,294,967,295
Ulong	Unsigned 64-bit integer, range 0 to 18,446,744,073,709,551,615
ushort	Unsigned 16-bit integer, range 0 to 65,535

Table A 2 - C# Types

A.3 Loops

Loops or iteration statements allow code to be executed several times. C# supports the following:

A.3.1 Do - While

The do-while statement executes a group of statements enclosed in {} repeatedly until a specified expression evaluates to false. The statement or block of statements will execute at least one time before the expression is tested.

```
public static void Main()
{

    int x = 0;

    do
    {
        Debug.Print("This is loop : " + x.ToString());
        x++;
    } while (x < 100);
}
```

A.3.2 While

The while-loop executes a group of statements enclosed in {} until a specified expression evaluates to false. The difference between the do-while loop and the while loop is that, the test expression is evaluated at the beginning of the loop for the while loop but it is evaluated at the end of the loop for the do-while loop.

```
public static void Main()
{

    int x = 0;

    while (x < 100)
    {
        Debug.Print("This is loop : " + x.ToString());
        x++;
    }
}
```

A.3.3 For

The for-loop executes a group of statements enclosed in {} repeatedly until a specified expression evaluates to false. The for-loop provides a mechanism for initializing variables before the loop is begun and a mechanism for modifying variables on each pass through the loop. This is handy for iterating over arrays and for sequential processing, and can also act as a delay.

```
public static void Main()
{

    for (int y = 0; y < 100; y++)
    {
        Debug.Print("This is loop : " + y.ToString());
    }
}
```

A.3.4 Foreach, in

The foreach,in statement looks through a collection to get specific information. It is not used for changing the contents in the collection.

```
public static void Main()
{

    int[] z = new int[] { 1, 4, 7, 12, 5 };
    foreach (int i in z)
    {
        Debug.Print("This is value : " + i.ToString());
    }
}
```

A.4 Program flow / Conditional Branching

Here are the program flow, otherwise known as conditional branching or selection statements that are available in C#:

A.4.1 If-else

The if-statement tests an expression for a Boolean result. If true the next statement in {} will be executed. The else is optional, and will be executed if the condition is false

```
public static void Main()
{

    if (w == 0)
    {
        Debug.Print("W is zero");
    }
    else
    {
        Debug.Print("W is not zero");
    }
}
```

A.4.2 Switch-Case

The switch statement is a control statement that evaluates a variable, in the switch statement, against a number of possible values, in the following case statements, and executes a specific block of code depending on the matching value. It also provides for a default block of code that can be executed if none of the cases are satisfied.

Example 1:

```
public static void Main()
{
    int x = 0;

    switch (x)
    {

        case 1:
        {
            Debug.Print("X is 1 ");
            Break;
        }
        case 2:
        {
            Debug.Print("X is 2 ");
            Break;
        }
        default:
        {
            Debug.Print("X is not 1 or 2");
        }
    }
}
```

Example 2:

```
public static void Main()
{

    String Test = "";
    switch(Test)
    {
        case 'A':
        {
            Debug.Print("Test is A ");
            break
        }
        case 'B':
        {
            Debug.Print("Test is B ");
            Break;
        }
        default:
        {
            Test = "C";
        }
    }

}
```

A.4.3 Goto

Before the advent of more structured programming languages, the only way to transfer execution control from one location in a program to another location based on runtime conditions was to use a goto statement. The goto statement transfers the program control directly to a labeled statement. There have been many a spirited discussion over the use of goto statements or the banning if them altogether. Statements have been made that claim that the number of bugs in a program are directly proportional the number of goto's in the program code. Nearly any programming structure that you might construct using goto statements can be alternately constructed using while, for, or do loops. There are, however, times when a goto is the most economic way to get the job done. We strongly recommend that you avoid using goto in your code, but if you feel you must, take a moment and write some meaningful comments with the goto to remind yourself later, when you are scratching your head in a debug session, just what you thought you were trying to accomplish by that goto statement.

```
public static void Main()
{

    if (x == 0)
    {
        x++;
        // Jumping to A because x is 0
        goto A;
    }
    else
    {
        // Jumping to B because x is NOT 0
        goto B;
    }

    A:
    Debug.Print("The value is " + x.ToString());

    B:
    Debug.Print("x is not zero");

}
```

A.4.4 Return

Return is used to send a value result back to an initiator for assignment. Here is a typical example:

```
public static void Main()
{
    String myResult = "";
    myResult = theTest(true);
}

public static string theTest(bool Test)
{

    if (Test)
    {
        return "The test was true";
    }
    else
    {
        return "The test was false";
    }
}
```

Although programmatically there is nothing wrong with the example above, having multiple return points can get confusing as the complexity of a method or sub-function increases. It is better programming practice to have a single point of return, if at all possible. It keeps you from returning uninitialized values and makes debugging much easier. A better structure for the same example would be:

```
public static void Main()
{
    String myResult = "";
    myResult = theTest(true);
}

public static string theTest(bool Test)
{
    // Return variable initialized with failure condition on entry
    String sReturnString = "The test was false";

    if (Test)
    {
        sReturnString = "The test was true";
    }

    // Single point of return
    return sReturnString;
}
```

A.5 Arrays

An array is a data structure (variable) that contains a number of variables of the same type. Arrays are declared with a type:

```
type[] arrayName;
```

Multidimensional arrays are not supported in .NET Micro Framework.

Create an integer array with no initialized values:

```
int[] x = new int[10];
```

Create an integer array with each element initialized to specific value:

```
        int[] y = new int[] { 0, 1, 4, 5, 6, 7 };
        int[] z = { 4, 7, 14, 13, 67 };
```

Notice that we had to specify the size of the array when we were not initializing the array elements; but when we provided initializers we did not have to specify the size. The compiler determines the size from the number of initializers in this case.

Create a string array:

```
        string[] myString = new string[4];
        string[] mystring = new string[] { "Test", "this", "array" };
```

Notice that this is an array of strings, and not a string represented by an array of characters.

C# also supports Arraylist, which provides better flexibility than the typical array. With Arraylists, you can add data, remove data, shuffle data, insert elements into an Arralylist, and you can get a count of the number of elements in the array.

B Virtual COM Ports

Virtual COM ports are used as a mechanism to provide driver access to hardware that is not directly supported by the CLR HAL/PAL API's. From the point of view of creating SerialPort objects and reading and writing data, programmatically they look identical to real COM ports. The differences are in the way data is managed so that the hardware that is associated with each virtual COM port can be properly configured and controlled.

When opening a virtual COM port, you specify a configuration that includes a baud rate. Baud rate really has no meaning for the ADC and PWM virtual COM ports, because in the iPac-9302, data is handled as parallel bytes and is never serialized. The choice of baud rate will not affect the transfer speed of either driver. One can choose any of the supported baud rates, and the virtual COM port will always run at its maximum speed. Similarly, flow control has no meaning, as well, and should be turned off.

The COM port interface in general has no clue about the particulars of the hardware device it is connected to. It knows how to send a byte-stream out and read a byte-stream in. Each virtual COM port driver has to add a command and control protocol on top of the byte-stream transport. The command and control protocol is a very simple structure that packetizes commands and data in a format that can be translated to appropriate register writes and reads of the target hardware device. The basic architecture of each command frame is show in Table B 1

Virtual COM Port Command Frame Architecture				
Byte 0	Byte 1	Byte 2	Byte 3	Byte 4…n
ASCII SOF	Command Code	Device Number	# of Data Bytes	Data Bytes

Table B 1 - Virtual COM Port Command Frame Architecture

Each frame starts with an ASCII Start-Of-Frame (SOC) byte (0x02) followed by a command code. Next is a device number, which for the current implementation can only be 0, but was included to support possible future hardware expansion. Next is the number of bytes that accompany the particular command. Some command codes require no data bytes, so this would be 0 and no bytes would follow. Some command codes require data to be sent, so the number of data bytes would be >0 and that would be the number of data bytes to follow and complete the frame. If the data that is being transported is of a data type that is more that 1 byte in size, the data is arranged in the frame as high-byte to low-byte order. This data exchange is all managed at a local level in the iPac-9302, so no frame check bytes are required. This keeps the data exchange lean and mean.

Command synchronization is important, so one of the commands supported is a synch command. If the software determines that it has gotten out of synch with the virtual COM port driver, it can issue a synch command and restart the command sequence.

This type of an interface is a little complex and not as intuitive as it would be if a direct API for the hardware device had been provided. This is where the strength of the managed code comes to our rescue, and we can wrap the complexities of the virtual COM interface in a managed code driver that provides a simple and intuitive API set for the software developer. Managed code drivers have been written for both of the virtual COM port devices. The fundamentals of the ADC and PWM virtual COM port drivers are described below:

B.1 SJJ_PWM_Driver Details

The following is a more in-depth description of the inner workings of the PWM managed code driver. As described earlier, this driver encapsulates the virtual COM port interface for the PWM controllers and provides a simpler and more intuitive API for the programmer.

If you review the source code for the PWM managed code driver, SJJ_PWM_Driver, you will see that it encapsulates the command definitions, status definitions, and byte arrays that are the frames required to write and read the registers of the PWM devices through the data transport of the virtual COM port. It also encapsulates the calculations that are required to determine some of the configuration register values, so that the software developer can concentrate on the frequency of the PWM output and the duty cycle as a percentage and not have to know how to convert those numbers into the appropriate PWM register values nor does the software developer need to know much about the hardware configuration, itself.

For those who may not want to use the managed code driver, let's look at one of the API methods to see how one goes about using the virtual COM port directly. A good example is the PWM_SetFreq() method:

```
public PWM_Status PWM_SetFreq(UInt32 uiPWMFreq, SerialPort myPwm)
{
    PWM_Status eReturnVal = PWM_Status.eStatusPWMComHandleError;
    UInt32 uiPWMxTermCnt;

    // Calculate terminal count
    uiPWMxTermCnt = (PWM_CLOCK_FREQ / uiPWMFreq) - 1;

    // Check that terminal count (and target frequency) is in range
    if (uiPWMxTermCnt <= PWM_MAX_TERMINAL_COUNT)
    {
        // Set terminal count in command string
        bSetFreq[bSetFreq.Length - 1] = (byte)(uiPWMxTermCnt &
0x000000FF);
        bSetFreq[bSetFreq.Length - 2] = (byte)((uiPWMxTermCnt >> 8) &
0x000000FF);

        // Write terminal count command to PWM virutal port
        if (myPwm != null)
        {
            byte[] bReadBuf = new byte[1];
            myPwm.Write(bSetFreq, 0, bSetFreq.Length);


            // Read and return status
            myPwm.Write(bSetStatusRead, 0, bSetStatusRead.Length);
            myPwm.Read(bReadBuf, 0, 1, 1);

            eReturnVal = (PWM_Status)bReadBuf[0];
        }
    }

    return (eReturnVal);
}
```

This method accepts the desired frequency, in Hz, and the SerialPort object that was returned by a previous call to the PWM_Open() method. If you look at the Cirrus EP9302 hardware reference, you will see that one needs to calculate the correct Terminal Count Register value and write it to that register. In the code above, you can see the calculation and then it is checked to be sure that the frequency that was requested was in a range that the PWM could actually provide. A check is made of the resultant Terminal Count Register value to be sure it does not exceed its range.

If the calculated register value is in range, it must be packaged in an appropriate command frame so that the virtual COM driver will write the new value to the Terminal Count Register. The virtual COM port driver has a command for writing the Terminal Count Register. In the managed code driver, that command is defined as `eWriteTermCnt` in the `PWM_Command` enumerated type. Also in the managed code driver, a byte array has been defined that has all the frame bytes to send this command to the virtual COM port driver with the data bytes for the Terminal Count Register value set to 0:

```
protected byte[] bSetFreq = new byte[] { (byte)PWM_Command.eSTX,
(byte)PWM_Command.eWriteTermCnt, 0, 2, 0x00, 0x00 };
```

The Terminal Count Register is a 16-bit register, so 2 data bytes are required. Going back to this method, the high byte and low byte of the calculated Terminal Count Register are written to the bSetFreq array's last two bytes by shifting and masking the `uiPWMxTermCnt` variable. Pay attention to the byte ordering when storing multi-byte variables. The bytes are written in highest byte to lowest byte order in the array.

With the register bytes written to the command frame array, it is ready to be sent through the virtual COM port driver. A sanity check it made to make sure that the SerialPort object is valid, and then the `myPwm.Write` operation is made with the `bSetFreq` command frame. The virtual COM port driver will parse the frame, recognize the command, pick off the register bytes, reconstruct a 16-bit integer with the data bytes, and write the new Terminal Count value to the Terminal Count Register.

To provide status, the virtual COM port driver registers a current driver status which can be read back with the `eSetReadReg` command, also found in the `PWM_Command` enumerated type. The data in the command tells the driver what to read back, so a single data byte of `eReadStatus` found in the `PWM_Read` enumerated type is included. The managed code driver has a frame already defined for that purpose:

```
protected byte[] bSetStatusRead = new byte[] { (byte)PWM_Command.eSTX,
(byte)PWM_Command.eSetReadReg, 0, 1, (byte)PWM_Read.eReadStatus };
```

Once this command is issued, the virtual COM port driver parses the frame, gets the current status, and puts it in the virtual COM port's read buffer. An immediate read operation, `myPwm.Read,` will return the status information that is then returned by the method.

If you review the other methods and definitions, you will get a clear idea of how the virtual COM port driver works for the PWM's and get an appreciation for how the managed code driver simplifies the interface.

B.2 SJJ_ADC_Driver Details

The following is a more in-depth description of the inner workings of the ADC managed code driver. As described earlier, this driver encapsulates the virtual COM port interface for the ADC controller and provides a simpler and more intuitive API for the programmer.

If you review the source code for the ADC managed code driver, SJJ_ADC_Driver, you will see that, as with the PWM managed code driver, it encapsulates the command definitions, status definitions, and byte arrays that are the frames required to write and read the registers of the ADC device. The virtual COM port driver for the ADC insulates the software developer from the specifics of the bit-fields in the ADC command registers, so even at this level, the software developer does not need to know the specific architecture of the ADC command and control registers.

For those who may not want to use the managed code driver, review the PWM managed code driver above, as both have the same command structure. Virtually all the configuration of the ADC is done through individual commands of the virtual COM port driver. These commands are defined in the `ADC_Command` enumerated type in the managed code driver.

Once the ADC is configured, an analog signal source selected, and the ADC turned on, sampling the selected source is similar to the reading back of the status. Let's look at the `ADC_Read` method:

```
public int ADC_Read(SerialPort myAdc)
{
    int iReturnVal = ADC_COM_ERROR;

    if (myAdc != null)
    {
        byte[] bReadBuf = new byte[2];

        myAdc.Write(bSampleADC, 0, bSampleADC.Length);
        myAdc.Read(bReadBuf, 0, bReadBuf.Length, 1);

        // Convert bytes to 16-bit integer
        iReturnVal = (int)bReadBuf[1];
        iReturnVal = (int)(iReturnVal << 8);
        iReturnVal |= (int)bReadBuf[0];
    }

    return (iReturnVal);
}
```

You will see that it takes the SerialPort object from the previous open operation as the only parameter. It is assumed that the ADC is configured, an analog channel selected, and the ADC has been turned on. The read method does a sanity check on the SerialPort object, and if valid, it writes a read request command frame in the form a defined byte array:

```
protected  byte[]  bSampleADC  =  new  byte[]  {  (byte)ADC_Command.eSTX,
(byte)ADC_Command.eSampleADC, 0, 0 };
```

This command is parsed by the ADC virtual COM port driver and the ADC is sampled and the results are placed in the virtual COM port's read buffer. This command is immediately followed by a read command, `myAdc.Read`, which returns the ADC value. The ADC return value is a 16-bit integer, so the read buffer is defined to be 2 bytes long, and then to return the 16-bit, signed integer, the returned bytes need to be shifted and OR'd into the return variable to be returned by the method. Note, again, the high-byte/low-byte ordering in the returned value.

Since we are returning the ADC result, either of the error conditions, ADC conversion error (returned by the virtual COM port driver) or NULL COM port handle, are indicated by returning predefined constants that are out of the valid ADC value range. These error return values are defined in the managed code driver as:

```
public const int ADC_CONVERSION_ERROR = 0x7000;  // Out of range value
to indicate conversion failure
public const int ADC_COM_ERROR = 0x7700; // Null COM port handle
public const int ADC_NEG_RANGE = 0x9E58;  // ADC count range is +/-
25,000 counts
public const int ADC_POS_RANGE = 0x61A8;
```

If you review the other methods and definitions, you will get a clear idea of how the virtual COM port driver works for the ADC device and get an appreciation for how the managed code driver simplifies the interface.

C Parts List, Suppliers, etc.

There are several extra parts that are used in the exercise that are not part of the EDK kit. Here is a list of parts and possible suppliers. You may find alternative sources in your local area or on the Internet.

C.1 Parts List

Part	Description	Supplier
NE555N	555 Timer	Jameco: 890091 / Digikey 296-1411-5-ND
ADC0804LCN	A/D Convert	Jameco: 10153 / Digikey ADC0804LCN-ND
Standoffs	13/16 inches (20.6 mm)	Radio Shack: 276-195
Bread Board	Bread Board	Jameco: 20601, 20758 / Digikey 438-1045-ND, 438-1046-ND
Jumper Wire	Jumper wire various lengths	Jameco: 19290, 20080 / Digikey 923351-ND
TIL311	HEX Display	Jameco: 32951
4-DIP switch	4-DIP switch	Jameco: 617908
Connection Wire	.025" square socket to .025 square socket	E-Z-Hook: 910-12-S
LCD2S-162 with 2x 16 (2 line x16 Character)	SPI LCD	Microconrollershop.com – LCD2S-162
4x4 keypad	Keypad to interface to SPI controller	Microconrollershop.com – HC-KP
Resistors: 1K, 10K 1K resistor pack	1K, 10K, and 1K resistor pack	
Capacitors: 10uF, 0.1uF, 150pF	10uF, 0.1uF, 150pF	
LEDs	2 –LEDs	
DC power Supply 0 to 15V, 0 to 3A or equivalent.	Optional external power supply	

Table C 1 - Exercise Parts List

C.2 Suppliers

The following are the supplier websites:

Company	Website
SJJ Embedded Micro Solutions	www.sjjmicro.com
Digi-Key	www.digikey.com
Radio Shack	www.radioshack.com
Jameco Electronics	www.jameco.com
E-Z-Hook	www.e-z-hook.com
Microcontrollershop.com	www.microcontrollershop.com

Table C 2 - Suppliers List

C.3 Schematic Capture

All the schematics in this guide were developed using Design Works Pro. The schematic and part library files are part of the EDK1.ZIP file. For more information about Design Works Pro and Design Works Lite, please visit Capilano Computing: http://www.capilano.com/.

C.4 Internet Resources

- .Net MF News Group: microsoft.public.dotnet.framework.microframework
- .NET MF website: http://msdn2.microsoft.com/en-us/embedded/bb267253.aspx

D iPac-9302 I/O Header Information

The iPac-9302 User Guide and the information listed in the GPIO section of this guide, detail the different GPIO ports available. This chapter is provided as a reference listing the pin header number and respective signals. The iPac-9302 has a arrow (▶) pointing to Pin1 on the PCB mask.

D.1 Header 1 – HDR1

Pin	Signal	Pin	Signal
1	PY7 (Output)	2	GND
3	PY6 (Output)	4	GND
5	PY5 (Output)	6	GND
7	PY4 (Output)	8	GND
9	PY3 (Output)	10	GND
11	PY2 (Output)	12	GND
13	PY1 (Output)	14	GND
15	PY0 (Output)	16	GND
17	PX7 (High Drive Output)	18	GND
19	PX6 (High Drive Output)	20	GND
21	PX5 (High Drive Output)	22	GND
23	PX4 (High Drive Output)	24	GND
25	PX3 (High Drive Output)	26	GND
27	PX2 (High Drive Output)	28	GND
29	PX1 (High Drive Output)	30	GND
31	PX0 (High Drive Output)	32	GND
33	PZ7 (Input)	34	GND
35	PZ6 (Input)	36	GND
37	PZ5 (Input)	38	GND
39	PZ4 (Input)	40	GND
41	PZ3 (Input)	42	GND
43	PZ2 (Input)	44	GND
45	PZ1 (Input)	46	GND
47	PZ0 (Input)	48	GND
49	(see JB1 options)	50	GND

Table D.1 – PLD-Based Digital I/O Connector (HDR1)

D.2 Header 2 – HDR2

Pin	Signal	Pin	Signal
1	ADC0	2	ADC1
3	ADC2	4	ADC3
5	ADC4	6	GND
7	GND	8	GND
9	INTR1	10	INTR3
11	PLDIO_00	12	PLDIO_01
13	PLDIO_02	14	PLDIO_03
15	PLD_PWM2	16	PLD_PWM3
17	GND	18	GND
19	EGPIO[13]/I2S-SDI2	20	EGPIO[14]/PWM1
21	GND	22	GND
23	SCLK1	24	SFRM1
25	SSPTX1	26	SSPRX1
27	GND	28	GND
29	ABTCK	30	ASYNC
31	ASDO	32	ASDI
33	ARSTN	34	EPGIO[4]/I2S-SDO1
35	EPGIO[5]/I2S-SDI1	36	EPGIO[6]/I2S-SDO2
37	GND	38	GND
39	5 Volts	40	3.3 Volts

Table D.2 - Analog (HDR2)

D.3 Header 3 – HDR3

Pin	Signal	Pin	Signal
1	HGPIO[2]	2	GND
3	HGPIO[3]	4	GND
5	HGPIO[4]	6	GND
7	HGPIO[5]	8	GND
9	FGPIO[1]	10	GND
11	CGPIO[0]	12	GND
13	EGPIO[1]/CLK_1HZ	14	GND
15	EGPIO[2]	16	GND
17	EGPIO[3]/HDLCLK1	18	GND
19	EGPIO[6]/I2S-SDO2	20	GND
21	EGPIO[10]DREQ1	22	GND
23	EGPIO[11]/DACK1	24	GND
25	EGPIO[12]/DEOT1	26	GND
27	EGPIO[13]/I2S-SDI2	28	GND
29	EGPIO[14]PWM1	30	GND
31	EGPIO[15]DASP	32	GND
33	PW0 (Input)	34	GND
35	PW1 (Input)	36	GND
37	PW2 (Input)	38	GND
39	PW3 (Input)	40	GND
41	PW4 (Input)	42	GND
43	PW5 (Input)	44	GND
45	PW6 (Input)	46	GND
47	PW7 (Input)	48	GND
49	3.3/5V (JB3)	50	GND

Table D.3 – Processor-Based Digital I/O Connector (HDR3)

E Extended Weak References

Extended Weak References was a stop gap solution for data storage for the early versions of .NET MF. The concept was interesting, but concern that the garbage collector could destroy data at any time is unacceptable to an embedded system. For completeness, we have moved the information about EWR to this appendix. The new FAT file system and SD card will be discussed in the future.

E.1 Memory Management (Garbage Collection) and Weak References

With a PC, you have different storage solutions available – CD-ROM, hard drive, floppy disk, USB flash key, compact flash card, SD card, etc. Although the iPac-9302 has a SD card slot, .NET MF didn't originally have support for it or even a file system available to write to it. The only storage available was the onboard flash. Without a file system, a special feature called Extended Weak Reference (EWR) was implemented to store data. In order to understand how EWR works, we need to understand something about the Garbage Collector (GC) and Weak References.

In C++, an object can be created and destroyed. Any object written in C++ should have a destructor in order to free-up memory. In C#, you create an instance of an object with the *new* operator, but there is no *delete* or *destroy* operator available. What frees the memory? The answer is the garbage collection unit. The garbage collector runs in the background cleaning up the memory heap of any unused objects. The garbage collector keeps track of the object references and looks for objects that can no longer be used by the running code. The garbage collector doesn't run constantly, so it doesn't take up valuable processor resources. The garbage collect runs periodically, ideally when the system is idle, but it is difficult to predict when it will actually run.

When you create an instance of an object using the *new* operator, this is called a strong reference. Weak References allow you to keep objects available, and they tell the GC to take memory if you have too. Normally, the GC cannot collect an object if it is reachable, i.e. if there is a direct or indirect reference to it. Such references are called strong references. A weak reference allows an object to be collected by the GC while it is still referenced. In such a case, the reference will either return a valid object or null if the GC has collected it.

In .NET Framework, Weak References can be used for large complex objects that take time to setup. If an application doesn't use the object for awhile but will come back to it, to save memory you can destroy the strong reference but keep a weak reference so you don't have to go through the setup again. If there is enough available storage space on the heap, the GC might not destroy the weak reference at all. It is a bit of a gamble, so you have to test to see if the weak reference is still alive before it can be used. Because of the small footprint, a large object in .NET MF would be rare, so you might not need to use weak references.

Information on weak references is limited. MSDN has some coverage, but the best resource is the book *Inside C#,* 2nd edition by MS Press. There is a weak reference example that demonstrates the concept. Here is the example adapted and modified for .NET MF and our purpose:

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

namespace SJJ_MF_Console_Application
{
    public class Program
    {
        public static void Main()
        {

            //Create a strong refernece
            myTest test1 = new myTest();

            //Create a weak reference
            WeakReference wr = new WeakReference(test1);

            //Test the weak reference
            if (wr.IsAlive)
            {
                myTest test2 = (myTest) wr.Target;
                Debug.Print("[Test2] " + test2.myString);
                test2.SaveString("Save this info");
                test2 = null;

            }

            //Destrong the Strong Reference
            test1 = null;

            //Force GC to run
            //Debug.GC(true);

            //Test that the Strong reference is gone
            //Debug.Print(test1.myString);

                //Do other Work

            for(int x = 0; x < 100; x++)
            {
                //Short delay
            }


            if (wr.IsAlive)
            {

                myTest test3 = (myTest)wr.Target;
                Debug.Print("[Test3] " + test3.myString);
                Debug.Print("[Test3] " + test3.myString2);
                test3 = null;
            }
            else
            {
                Debug.Print("[Test3] WeakReference is Dead");
            }

        }

    }


    public class myTest
    {
        public string myString = "Testing Weak References";
        public string myString2;

        public void SaveString(string myString2Input)
        {
            myString2 = myString2Input;
        }
    }

}
```

Weak references can only be accessed via a static method, so we deviate from our App class. Test1 is a strong reference to the myTest class. A weak reference “wr” is created for the instance of Test1. The application tests that the weak reference is alive. If alive, test2 is used to create an instance of the weak reference, performs some actions, and is then destroyed.

We then destroy the strong reference Test1 (Test1 = null;). The application can then go off and do some work, and then the weak reference is tested again. If it is still alive, Test3 becomes an instance of the weak reference, and Test3 is able to perform some actions. The output is:

```
[Test2] Testing Weak References
[Test3] Testing Weak References
[Test3] Save this info
```

If the Debug.GC(True) line is uncommented, the output would be:

```
[Test2] Testing Weak References
[Test3] WeakReference is Dead
```

The GC destroyed the weak reference so it cannot be used again. If you were to comment out the Test1 = null line so the strong reference is not destroyed, the weak reference would not be destroyed.

If you were to uncomment the Debug.Print(test1.myString) when Test1 is destroyed, the application would crash since Test1 no longer exists.

The example shows the GC in action, and a valuable concept about keeping objects alive with the GC running in the background.

E.2 Data Storage with Extended Weak References

The previous section focuses in on an important issue of the GC. It may seem like a long way to go to talk about data storage, but when you are working with data that is going to be stored in flash, you need to be aware that the GC is running in the background.

ExtendedWeakReferences (EWR) are unique to .NET MF that allow you to take an object and store it in flash. The internal flash driver is use to write the data. EWR is part of the Microsoft.SPOT namespace.

The actual object itself must be flagged as serializable. Serialization is a process for converting an instance of an object to a format that can be saved to a file, database, or transmitted across the network. Web applications take advantage of serialization. In .NET MF, we want to store the object into flash. In order to define a EWR of an object, the object itself must be serializable.

E.3 EWR Class

EWR consists of an EWR class and defined priority level enumeration. The constructor ExtendedWeakReference initializes a new instance of EWR that will be used to reference an object. The following are the methods available:

- RecoverOrCreate – Attempts to recover a specific EWR object, and if it is not recoverable it will create a new instance of the class.
- Recover – The object is known to exist, so just recover the object to access the data.
- PushBackIntoRecoverList – Flags a EWF object for recovery in the event of a power cycle.

Since the PC has been around for some time, everyone is used to seeing files with names and files in directories. In .NET MF only one application can be running at a time, but the application could access multiple objects stored in flash. Your application is going to be performing and managing the file reads / writes.

The RecoverOrCreate method is used to create a new EWR object that will be stored in flash. The input parameters are as follows:

- Selector – This is a type that is associated with the EWR you want to recover or create. In other words think of this as your file name.
- ID – The ID of the EWR you want to recover. Another way to think about this is a page in the file you want to recover. You can have multiple pages in a file, if you want to preserve the data and start new, you can just set a new page number.
- Flags – If the EWR object is new, the flag defines how the data will be treated in a reboot. There are two options – c_SurviveBoot (survive a reset of the system) and C_SurvivePowerDown (shutdown or power down).

The Recover method only requires the Selector and ID to recover the data. In theory, you can have one application create and write some records to flash and then download a new application that just recovers the data.

Once you have the EWR object, you need to tell the CLR how to treat the data. This is where the GC concepts in the last section become important. With EWR, you can tell the GC the importance of the data. The GC will determine what data to discard based on priority going from OkayToThrowAway up to System items. The table describes the different priority levels.

Priority Level	Description
System	System data that is mandatory for basic operation
Critical	Critical data.
Important	Data that is important, but not critical.
NiceToHave	Data that is nice to have, but not critical.
OkayToThrowAway	Data that can be safely thrown away.

Fig 11.1 - EWR Priority Levels

It is important to note that there is no concept of getting available storage information. As flash fills up with data, the CLR makes a decision on what data to keep or save based on the priority level set.

E.4 A Simple EWR Example

Okay, now let's save some data. The SDK comes with a sample EWR application for a custom emulator. The emulator that is part of the project has an added feature to store data in a file on a local hard drive. The application itself can be made to run on the iPac-9302. The EWR project in the Appendix E folder is a modified version of the SDK example. The application simply counts the number of boots.

Since our philosophy is that the more examples available the better, for this exercise, we will create an application from scratch. The application will still count boots, but it will also store a string of data.

E.4.1 Create the Storage Application

1. Open Visual Studio or Visual C# Express Edition.

2. Using the SJJ_MF Console Application, create a new application called Storage_EWR1. There is no need to add the SJJ Hardware provider since GPIOs, SPI, or COM ports will not be used.
3. In the App class, and before the Run() method, add the following:

```
private static ExtendedWeakReference myData;

private static class TypeAppDataFlag { }
```

The first line defines the ExtendedWeakReference. The second line is a class object that will be used as our file name. When you want to access the data in the future, your application would have to use the same class. The class is case sensitive.

4. Now let's add what we want to actually store. After the two lines just entered add the following:

```
[Serializable]
private sealed class theData
{

    public Int32 BootCount;
    public String theDataString;

    public void newData(string theString)
    {
        theDataString = theString;
    }

    public void newBootCount(Int32 bootCount)
    {
        BootCount = bootCount;
    }
}
```

The Serializable attribute is required so the data can be saved to flash. The class itself is a sealed class, meaning that class is inheritable for use with other classes. The actual data itself is simply two items: a 32-bit integer for holding the number of boots and a string. We can store other information such as other strings, integer values, arraylists, etc., but we want to keep this as simple as possible.

5. Right after our theData class, lets create a new reference to the class:

```
private static theData myStoredData;
```

6. Now we are ready to create the retrieve and save methods. Add the following:

```
public static void GetData()
{

    myData = ExtendedWeakReference.RecoverOrCreate(typeof(TypeAppDataFlag), 0,
    ExtendedWeakReference.c_SurvivePowerdown);
    myData.Priority = (Int32)ExtendedWeakReference.PriorityLevel.Important;

    myStoredData = myData.Target as theData;

    if (myStoredData == null)
    {
        Debug.Print("First Boot or data was lost");
        myStoredData = new theData();
        myStoredData.newBootCount(0);
        myStoredData.newData("New");

    }

}
```

```
public static void SaveData()
{
    myData.Target = myStoredData;
    Thread.Sleep(4000);
}
```

In the GetData method, the EWR myData attempts to recover the data on the flash disk using the TypeAppDataFlag object and index. If the information doesn't exist, it will created it with the attributes to survive a power cycle and the second line lets the GC know the importance of the data.

The third line associates our data class with the EWR. The 'as' reference converts the data, if any, to the defined data class myStoredData. If the data doesn't exist, the 'if' condition will use the methods to add data.

The SaveData method simply passes the object, myStoredData, to the EWR myData. This will trigger the data to be written to flash. The Thread.sleep delays any further execution so data has time to be written to the flash.

Notice that all of the GetData and SaveData are 'static'. EWR can only be accessed using Static methods. Static methods cannot access non-static methods or properties, but non-static methods can access static methods and properties. We can use the Run method created by the template to perform the restore and save actions.

7. Finally, lets add code to our Run method:

```
//Load the old data from flash
GetData();

Debug.Print("Number of boots is " + myStoredData.BootCount);
Debug.Print(myStoredData.theDataString);

//Manipulate the data
myStoredData.BootCount++;
if (myStoredData.theDataString == "String A")
{
    myStoredData.newData("String B");
}
else if (myStoredData.theDataString == "String B" || myStoredData.theDataString ==
"New")
{
    myStoredData.newData("String A");
}

Debug.Print("The new data to be saved:");
Debug.Print("Boot number is " + myStoredData.BootCount);
Debug.Print(myStoredData.theDataString);


//Save the updates
SaveData();
Debug.Print("Data Saved!");
```

The first thing is to retrieve the stored data. If the data doesn't exist, than GetData will create the data. The currently stored data will be sent to the debug output. The ability to manipulate and update stored data could be important to a project, so the boot count is incremented and a new string is stored. The udpated data is then sent to the debug output, and the data is saved to flash. The application then exits.

8. Save the project.

E.4.2 Build, Deploy, and Test

1. Make sure the Null modem cable is connected to the iPac-9302 and the development computer.
2. With the project selected in the Solution Explorer, from the menu, select **Project** and then select **Storage_EWR1 Properties…** from the submenu.
3. The project properties page appears, click on the **Micro Framework** tab.
4. Under Deployment, select **Serial** for the Transport and **Com#** as the Device. Where # is the COM port number of the development computer.
5. Save the project
6. From the menu, select **Build**, and then select **Build Storage_EWR1** from the drop down menu.
7. Make sure the iPac-9302 is powered on. You need to wait at least 20 seconds before deploying an application. This gives TinyBooter a chance to turn control over to the CLR. You can also monitor the boot-up of the iPac-9302 with MFDeploy and use the clearing of the BootLoader Flag to shorten the boot time, but **remember to disconnect MFDeploy before you attempt to deploy an application from Visual Studio.**
8. From the menu, select **Build**, and then select **Deploy Storage_EWR1**.
9. The output window should show a successful download.
10. Launch a terminal application like SJJ_COMM to view the debug output.
11. Power-cycle the iPac-9302 and watch the output. The first time the application runs the data doesn't exist so the output should look like the following:

```
Hello World!
First Boot or data was lost
Number of boots is 0
New
The new data to be saved:
Boot number is 1
String A
Data Saved!
Done.
```

12. Hit the reset button, and the data gets updated:

```
Hello World!
Number of boots is 1
String A
The new data to be saved:
Boot number is 2
String B
Data Saved!
Done.
```

If you continue to hit reset, you will see the boot count increase and the string value ping-pong between String A and String B. You can manipulate theData class to store other information. One item that might be interesting to store is an Arraylist. You can use an Arraylist to keep a log of records, but you need to be careful in increasing the log. The .NET MF port to the iPac-9302 supports up to 1MB of storage data. You will have to manage storage in your application and delete old data when necessary.

F Bibliography

Various resources were used to develop this guide. There are many in-depth resources on C# development.

F.1 Books:

C# Programmer's Cookbook, Allen Jones, Microsoft press, 2004, ISBN: 0-7356-1930-1

CLR via C#, Jeffrey Richter, Microsoft Press, 2nd Edition, 2006, ISBN-13: 978-7356-2163-3, ISBN-10:0-7356-2163-2

Embedded Programming with the Microsoft .NET Micro Framework, Donald Thompson and Rob S. Miles, Microsoft Press, 2007, ISBN-13: 978-0-7356-2365-1, ISBN-10: 0-7356-2365-1

Inside C#, Tom Archer and Andrew Whitechapel, Microsoft Press, 2nd Edition, 2002, ISBN: 0-7356-1648-5

Introducing .NET, David S. Platt, Microsoft Press, 3rd Edition, 2003, ISBN: 0-7356-1918-2.

Microsoft Visual C# .NET (Core Reference), Mickey Williams, Microsoft Press, 2002, ISBN: 0-7356-1290-0.

Windows XP Under the Hood, Brain Knittel, QUE, 2003, ISBN: 0-7897-2733-1

Visual C# 2008 Step-by-Step, John Sharp, Microsoft Press, 2007, ISBN: 0-7356-2430-5

F.2 Conferences

Instructor-Lead Lab: ILL316 - Customizing the .NET Micro Framework Emulation System to Simulate Real Hardware, MEDC 2007

F.3 Websites

History of SPI and the 555 Timer were found here:
http://en.wikipedia.org/wiki/Serial_Peripheral_Interface_Bus
http://en.wikipedia.org/wiki/555_timer

History of Ethernet:
http://www.youtube.com/watch?v=g5MezxMcRmk

Tera Term Download Site:
http://logmett.com/index.php?/download/tera-term-464.html