What is an Arduino shield and why should my Netduino care?

C9 Netduino Shield Series - Using Arduino Shield with Netduino - Part I

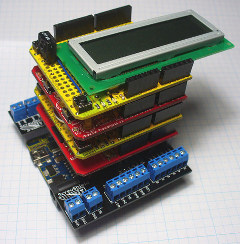


Photo courtesy of [John Boxall](http://tronixstuff.wordpress.com/)

# Introduction

When the [Arduino](http://www.arduino.cc/en/Guide/Introduction) Duemilanove microcontroller appeared in 2005, it featured a set of female pin headers exposing most of the pins of the ATmega168 for easy hacking and for connecting accessory boards known as 'Shields'. The purpose of a shield is to provide new plug-and-play functionality to the host microcontroller, such as circuit prototyping, motion control, sensor integration, network and radio communication, or gaming interfaces, without worrying too much about the hardware implementation details. Seven years after the birth of the original Arduino, new shields keep coming out and are being cataloged on <http://shieldlist.org/>, a testament to the versatility of the design. It is also simple to build a DIY shield when nothing out there will meet your needs or when you want to understand how the shield concept works from the ground up.

In their infinite wisdom, Secret Labs, the makers of the [Netduino](http://www.netduino.com/), adopted the same Arduino form factor and header pin out to let .Net Micro Framework users tap into the vast array of Arduino shields on the market. Yet, there's always one hurdle that needs to be overcome: in order to give access to their hardware functions easily, many Arduino shields provide [sample sketches](http://www.arduino.cc/playground/Main/GeneralCodeLibrary) (a.k.a. C/C++ [Wiring programs](http://wiring.org.co/)) and [libraries](http://arduino.cc/en/Reference/Libraries) also [written in C/C++](http://arduino.cc/en/Reference/HomePage) designed to be used within the [Arduino IDE](http://arduino.cc/en/Guide/Windows" \l "toc6). Unfortunately, Arduino sketches and libraries aren't compatible with the Visual Studio environment and the .Net runtime: they need to be adapted in order to make use of the shield's hardware.

The goal of this series of article is to attempt to demystify what an Arduino shield is. To do this, we'll start by building one from scratch. Then, we'll present a generic approach for selecting Netduino-compatible Arduino shields and adapting Arduino code to interface with the .Net Micro Framework.

### Selecting Arduino shields for use with Netduino / Plus

Arduino shields using hardware components interfaced over protocols such as [SPI](http://en.wikipedia.org/wiki/Serial_Peripheral_Interface_Bus), [I2C](http://www.i2c-bus.org/), [1-Wire](http://www.maxim-ic.com/products/1-wire/flash/overview/index.cfm) or serially over a [UART](http://en.wikipedia.org/wiki/Universal_asynchronous_receiver/transmitter) are generally good candidates for a Netduino adaptation because the .Net Micro Framework supports these protocols natively and can do bulk I/O transfers, yielding good performance.

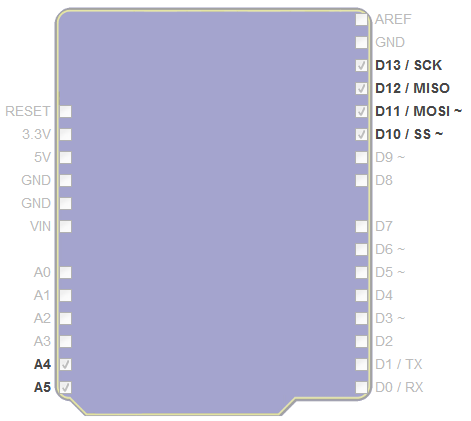
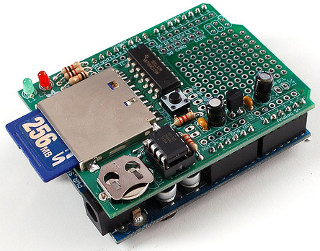
However, if a shield requires [bit-banging](http://en.wikipedia.org/wiki/Bit_banging) data on GPIOs (i.e. communicating data serially, such as SPI, I2C, RS-232, in software by toggling GPIO lines individually instead of using dedicated hardware interfaces) or has very low I/O latency requirements, it may not be practical or even possible to use the shield on a Netduino / Plus.

The reason for this is simple: the Arduino runs native code and can access hardware registers directly with a latency orders of magnitude lower than what the .Net Micro Framework can achieve on a Netduino / Plus where toggling GPIOs means going through multiple framework layers while running interpreted code. The maximum raw I/O toggling frequency on a Netduino / Plus clocked at 48 MHz [was measured around ~8.4KHz](http://fabienroyer.wordpress.com/2011/01/22/netduino-gpio-speed-test/) which is slow compared to the few MHz achievable on an Arduino clocked at 16 MHz.

Fortunately, there are possible workarounds and strategies

* Whenever an Arduino library bit-bangs SPI data on GPIOs, regardless of the pins used to do it, it can be replaced by using the standard SPI interface on the Netduino / Plus.
* The Netduino / Plus has significantly more RAM available than the Arduino and can transfer larger SPI data buffer in a single shot, dramatically reducing I/O latency. For instance, [driving a display shield would greatly benefit from this method](http://fabienroyer.wordpress.com/2011/09/14/driving-an-adafruit-st7565-negative-lcd-display-with-a-netduino/) where a frame is cached in RAM on the Netduino / Plus then sent in one shot over SPI to the hardware display driver.

The bottom line is that it is always a good idea to review how the Arduino code interfaces with a given shield before proceeding with a purchase or starting a conversion project. This review process is also facilitated by [http://shieldlist.org](http://shieldlist.org/) which shows which pins are being used to drive a shield. For example, the [Adafruit Industries Logger Shield](http://shieldlist.org/adafruit/logger) uses hardware SPI (pins D10-D13) for the SD card and I2C



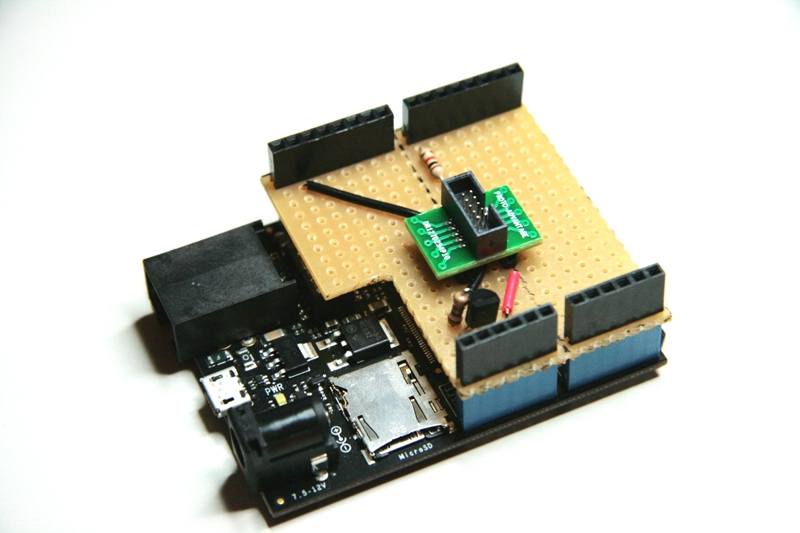
(pins A4-A5) for the DS1307 real time clock, making it a perfect adaptation candidate for the Netduino / Plus as we'll see in our next article.

It's also very important to know what features are available on the .Net Micro Framework running on the Netduino / Plus: some things come standard with the .Net Micro Framework which normally require add-ons libraries on the Arduino such as file system and networking support.

Finally, it is wise to search code repositories such as [CodePlex](http://www.codeplex.com/), [BitBucket](https://bitbucket.org/) and [GitHub](https://github.com/) for hardware drivers and libraries before undertaking writing one in C# from scratch based on component datasheets. This extends to the [Netduino](http://forums.netduino.com/index.php?/forum/15-project-showcase/) and the [TinyCLR](http://www.tinyclr.com/forum) community forums where many people have posted drivers in discussion threads.

## Building a shield from scratch to meet specific requirements

Recently, we were asked if it was possible to connect our [Touch Display module](http://nwazet.com/touch-display-module) designed for a [Netduino Go!](http://www.netduino.com/netduinogo/specs.htm) to a [Netduino](http://www.netduino.com/netduino/specs.htm) / [Plus](http://www.netduino.com/netduinoplus/specs.htm). By doing a bit of research and seeing how the driver for the Touch Module works, this can be done by building a simple custom shield composed of a few parts connecting the Netduino pin headers to a Netduino Go! socket breakout board like so:



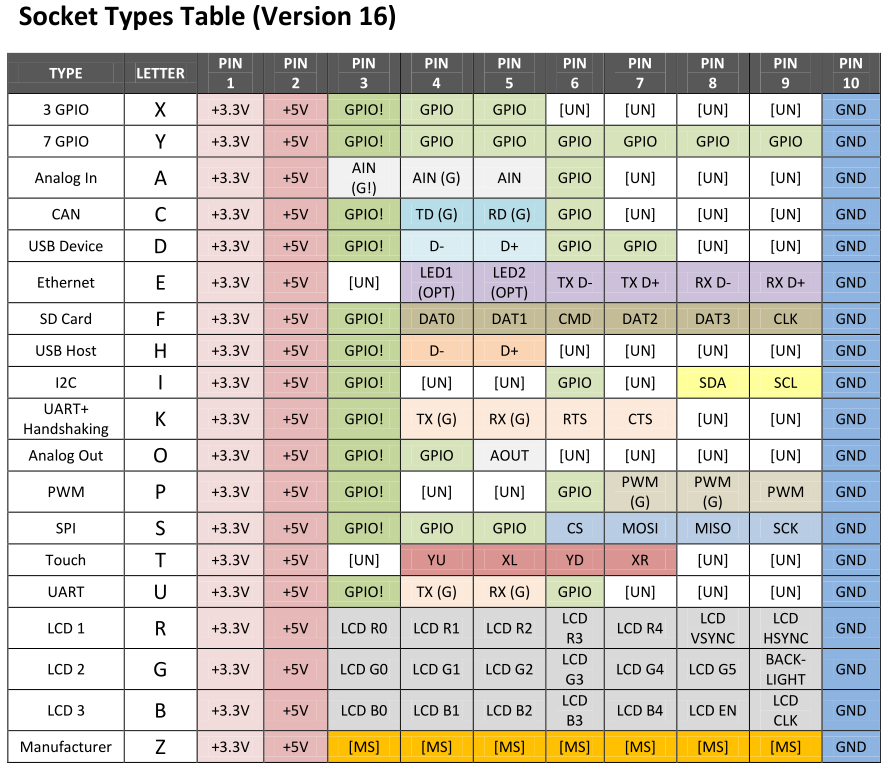


### Netduino Go! Socket Pin Out

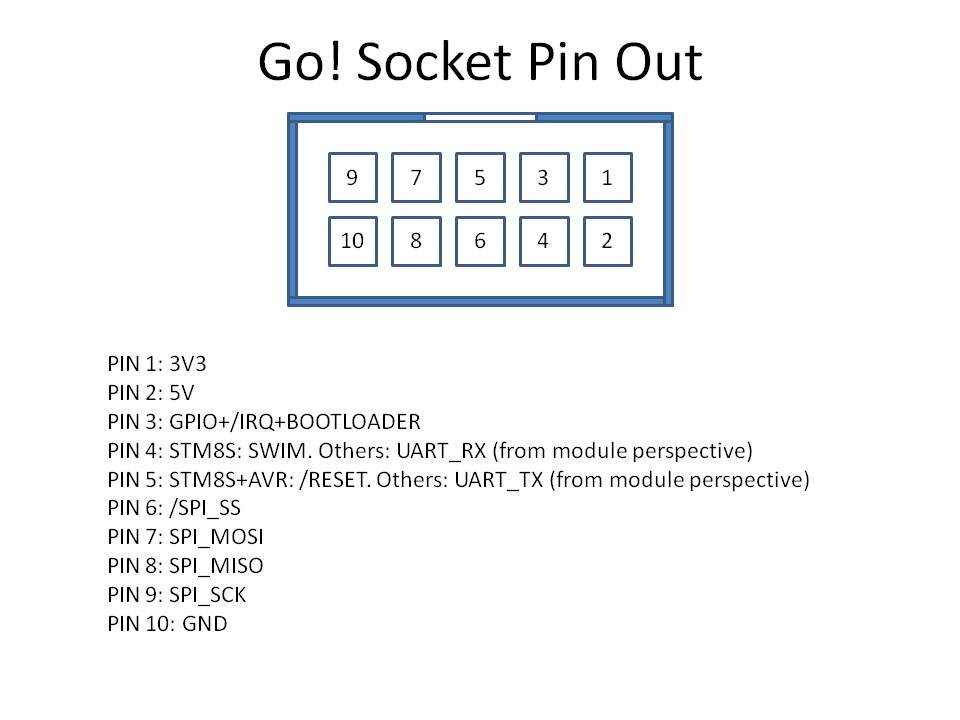
In order to build the shield, we need to start with mapping out the connections between the Go! bus socket and the header pins of the shield.

The Go! socket pin out was designed by Secret Labs to be compatible with [Gadgeteer modules supporting the following socket types](http://gadgeteer.codeplex.com/releases/view/72208): S (SPI), U (UART), X (3 GPIO) as documented in the [.Net Gadgeteer Mainboard Builder's Guide v 1.8, page 6](http://gadgeteer.codeplex.com/downloads/get/273357). This makes our little shield compatible with a Gadgeteer board as long as it is connected to an S socket. As far as other .Net Micro Framework boards are concerned, they should also work fine if they can drive an SPI slave device at 5 MHz, the minimum SPI clock frequency required by the Touch Display module.

#### Gadgeteer Socket Types



#### **Netduino Go! Socket**

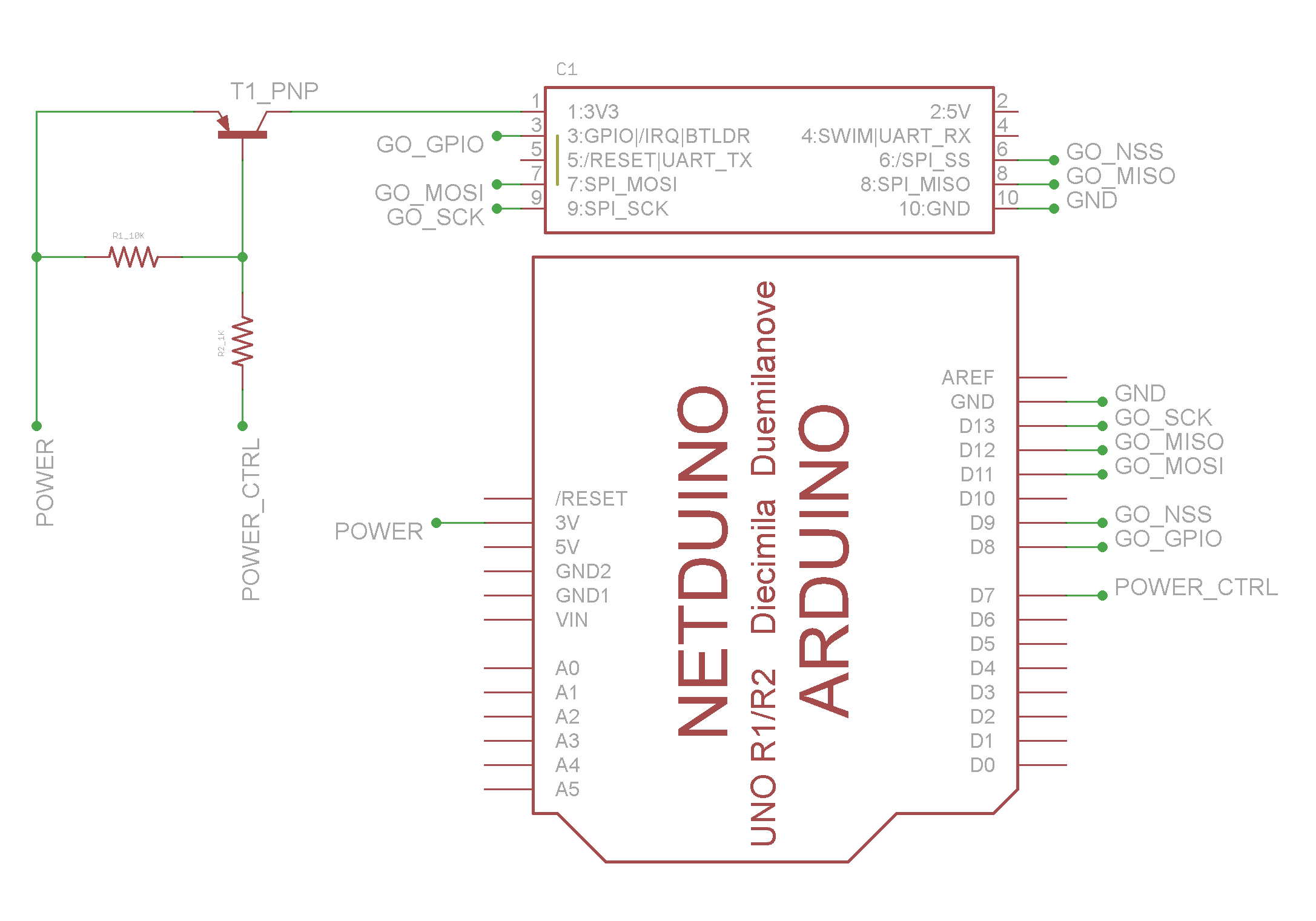


The [hardware specifications of the Netduino](http://www.netduino.com/netduino/specs.htm) / Plus tells us that the Netduino GPIOs correspond exactly to the [Arduino Uno GPIOs](http://arduino.cc/en/Main/ArduinoBoardUno):

* digital pins 0-1: UART 1 RX, TX
* digital pins 2-3: UART 2 RX, TX
* digital pins 5-6: PWM, PWM
* digital pins 7-8: UART 2 RTS, CTS
* digital pins 9-10: PWM, PWM
* digital pins 11-13: SPI MOSI, MISO, SPCK
* analog pins 4-5: I2C SDA, SCL

With this information, we can proceed with the creation of schematics to bridge the gap between the two worlds.

### Shield schematics



### How does this custom shield work?

The Netduino Go! bus is designed around the [SPI protocol](http://en.wikipedia.org/wiki/Serial_Peripheral_Interface_Bus) (see Go! socket pins 6-10) which the Netduino / Plus can speak easily, making a Netduino Go! module or a [S-type Gadgeteer module](http://www.ghielectronics.com/catalog/category/275/) appear like any other SPI device to a Netduino / Plus.

You will notice that the following pins aren't connected on the Go! socket for the sake of simplification:

* Pin 2 (5v power): the Touch Display module operates on 3.3v, therefore the 5v power supply is not needed but could be connected to the 5v header of the Netduino to be fully compliant.
* Pin 4 and 5 (UART): the Touch Display module does not make use of the serial interface during normal operation, so these lines can remain disconnected. However, to be fully compliant, Pin 4 (RX) and 5 (TX) on the Go! socket could be respectively connected to Netduino pin D1 (TX) and D0 (RX).
* The fact that these lines aren't connected in this specific scenario would cause another module requiring a UART and/or 5v to not work obviously.

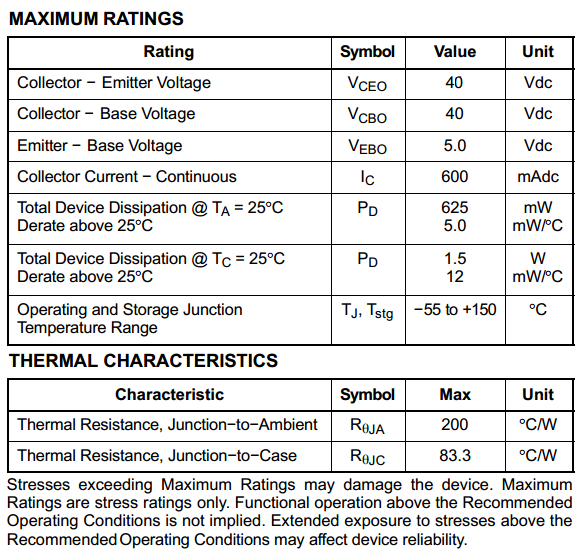
While the SPI part is straight forward, there's a bit more to the Go! bus electrical specification that must be taken into account: the Netduino Go! bus also uses its GPIO pin on boot (see Go! socket pin 3) to control the behavior of modules based on ARM Cortex chips: when the GPIO pin is held HIGH on module power-up, the ARM Cortex chip enters its bootloader mode and waits to be flashed with new firmware over the Go! bus UART (see Go! socket pins 4-5). The ARM Cortex bootloader remains active until the ARM Cortex chip is reset or power-cycled.

This is important to know for two reasons:

1. The Touch Display module is powered by an ARM Cortex M3 chip.
2. On boot, a Netduino / Plus sets all of its I/O pins HIGH, preventing the Touch Display from booting its firmware, which is undesirable. The reason for the I/O pins being set HIGH is documented on [page 5 of the ATMEL AT91SAM7X datasheet](http://www.atmel.com/Images/doc6120.pdf) which states that the pins on Parallel IO Controller A and B (PA0 - PA30 and PB0 - PB30) are pulled-up input at reset. The pins stay in that HIGH state until they're reconfigured by the Tiny CLR and the C# application itself.

For this reason, the custom shield is designed to control the power supply of the Touch Display module through a general-purpose [2N4403](http://www.onsemi.com/pub_link/Collateral/2N4403-D.PDF) [PNP transistor](http://www.onsemi.com/pub_link/Collateral/2N4403-D.PDF) (see T1\_PNP on the shield schematics) connected to pin D7: it is only when the base of the transistor is asserted LOW that power flows to the Touch Display module, countering the behavior of the Netduino / Plus on boot. In addition, the Netduino / Plus controls the state of the Go! bus GPIO pin through D8, asserting it LOW before powering up the Touch Display module.

About the 2N4403 transistor: any other PNP transistor, or a [P-Channel MOSFET](http://www.electronics-tutorials.ws/transistor/tran_6.html), capable of switching at least 150mA continuously, the maximum current consumption of the touch display with some headroom, would have worked equally as well. The 2N4403 used here is more than adequate:



In the context of this shield, the PNP transistor is used as a switch, either fully ON or OFF, as opposed to being used as an amplifier. To this end, a 1K ohm resistor is used between the base of the transistor and Netduino / Plus pin D7 (POWER\_CTRL) to ensure that the transistor is fully saturated but also limiting the maximum current sunken through pin D7 and potential damage being done to the ATMEL chip. The 10K ohm pull-up resistor placed between the base of the transistor and the 3.3v power rail ensures that the transistor will always be turned off by default, even when pin D7 is floating (i.e. not specifically declared as an Input or as an Output and driven HIGH or LOW by the C# application).

The PowerUpDisplay function summarizes the interaction with the module through the shield on boot:

// Pin connected to the transistor controlling the display's power supply

public static OutputPort PowerTransistor = new OutputPort(Pins.GPIO\_PIN\_D7, true);

public static void PowerUpDisplay() {

// Ensure that the GPIO pin in low to prevent the display module to start in bootloader mode

var goBusGPIO = new OutputPort(Pins.GPIO\_PIN\_D8, false);

// Power up the display module

PowerTransistor.Write(false);

// Always wait 250ms after power-up to ensure that the display module is fully initialized before sending commands

Thread.Sleep(250);

goBusGPIO.Dispose();

}

### Refactoring a Netduino Go! C# driver to work on a regular Netduino / Plus

The final step to get the Touch Display module working outside of its natural environment involved refactoring its [C# driver (VirtualCanvas.cs)](https://bitbucket.org/fabienroyer/nwazet.go/src/b24909aff18e/Library/Nwazet.Go.SDT028ATFTLCDTS/Nwazet.Go.SDT028ATFTLCDTS/Imaging/VirtualCanvas.cs), eliminating the idiosyncrasies specific to Netduino Go! bus. [Go! modules are supposed to implement the *GoModule* interface as defined by the Netduino Go! SDK](http://www.netduino.com/downloads/). Because the regular Netduino / Plus SDK has no knowledge of this interface, it's necessary to strip out any references and API calls specific to the Go! framework, leaving only what's absolutely required to communicate with the Touch Display module over SPI and maintaining the disposable nature of the class. For more details on Go! SDK, please refer to [the Netduino Go! community forums](http://forums.netduino.com/index.php?/forum/26-netduino-go/) and [the source code of the existing module drivers](http://wiki.netduino.com/List-of-all-GO-Modules.ashx) as it is currently the best source of documentation.

This step is akin to the work required when adapting an Arduino library to run on the .Net Micro Framework, only in this case, the task is easier since we're dealing with a C# driver to begin with instead of going from C/C++ to C#.

Starting from the top of VirtualCanvas.cs, the following changes were made in the Netduino / Plus version of the driver:

* Removed
  + *using GoBus;*
* Replaced
  + *public class VirtualCanvas : GoModule*
  + *with: public class VirtualCanvas : IDisposable*
* Replaced
  + *public void Initialize(GoSocket socket, uint speedKHz = 25000)*
  + *with: public void Initialize(SPI.SPI\_module displaySpi, Cpu.Pin displayChipSelect, Cpu.Pin displayGPIO, uint speedKHz = 25000). The core of the function simply initializes the SPI interface.*
* Replaced
  + *protected override void Dispose(bool disposing)*
  + *with: public void Dispose() also removing the following calls:*
    - *SetSocketPowerState(false); this call is used by the GoModule object model to control when a socket is powered ON / OFF and is no longer applicable when the VirtualCanvas class is not derived from GoModule.*
    - *base.Dispose(disposing); by the same token, deriving from GoModule requires overriding the behavior of the Dispose function and calling the base class at the end, something that is no longer needed now that the class implements the regular IDisposable interface.*
* Moved
  + *GoBusIrqEvent.Reset(); from WaitUntilGoBusIrqIsAsserted()*
  + *to: Execute(Synchronicity sync = Synchronicity.Synchronous) taking into account the higher latency of the Netduino / Plus*

The rest of the code remained unchanged. The Netduino / Plus driver and test application for this custom shield can be downloaded from https://github.com/fabienroyer/Channel9. In addition, the following video shows the Nwazet Touch Display module and the shield in action with a Netduino Plus: https://www.youtube.com/watch?v=m0-RUMYaZEY

## Conclusion

With nearly 300 known Arduino shields on the market, there's a wide variety of plug-and-play functionality waiting to be leveraged by the Netduino community, sometime with little or no software work required. When all else fails, building a shield from scratch to get a desired feature onto a Netduino / Plus is a relatively simple process only requiring reading component datasheets and some patience with a soldering iron or a small breadboard. In our next article, we'll put these guidelines to the test with a simple and popular off-the-shelf data logger shield made by Adafruit Industries.

## Appendix

To build the shield interfacing with the Nwazet Touch Display, you will need the following parts:

### Custom shield Eagle schematics

* Custom shield : https://github.com/fabienroyer/Channel9

### Bill of materials (~$15 for the shield w/o the Touch Display module)

* 1 [[nwazet Touch Display module](http://nwazet.com/touch-display-module)
* 1 [10cm FCI connection cable](http://nwazet.com/gocable10)
* 1 [Adafruit DIY shield for Arduino](https://www.adafruit.com/products/187)
* 1 [ProtoAdvantage IDC Cable Breakout Board](http://www.proto-advantage.com/store/p/Netduino_GO_IDC_Adapter.php)
* 1 [PNP Transistor 2N4403](http://components.arrow.com/part/detail/40492237S7076394N5236)
* 1 [1K ohm resistor](http://components.arrow.com/part/detail/39827066S7077451N5146)
* 1 [10K ohm resistor](http://www.digikey.com/product-detail/en/CF14JT10K0/CF14JT10K0TR-ND/1741265?cur=USD)
* 1 [22 AWG solid core wire](http://www.radioshack.com/product/index.jsp?productId=2049745)
* 1 [0.32" diameter 60/40 rosin-core solder](http://www.radioshack.com/product/index.jsp?productId=2062725)

### Eagle libraries

* [Adafruit Eagle library](http://www.ladyada.net/library/pcb/eaglelibrary.html) (Netduino shield part)
* [[nwazet Eagle library](https://bitbucket.org/fabienroyer/nwazet.go/src) (Go Socket part)

### Netduino Go! Wiki

* [Netduino GO! Module Builder's Guide](http://wiki.netduino.com/Netduino-GO-Module-Builders-Guide.ashx) (documents the Go! bus specifications)