adafruit learning system

Using NeoPixels with Netduino Plus 2

Created by Frank Zhao

Last updated on 2018-08-22 03:37:14 PM UTC

Guide Contents

Overview

Good news for Netduino users who have been eyeing the glamorous NeoPixels strips and shapes in the Adafruit shop! We have managed to get NeoPixels working on the Netduino Plus 2 (shown below with a NeoPixel 8x8)

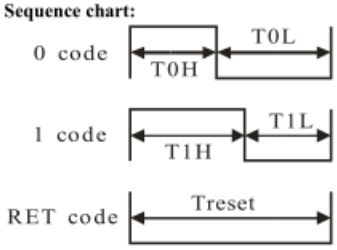
Even though there is lots of example code for NeoPixels for AVR, Propeller and PIC chips, it's a bit of challenge getting these LEDs to work with the Netduino. With the default firmware of the Netduino (any variation, Plus or no), it is not possible to send data to NeoPixels.

The reason is due to the special high-timing-specific nature of the data sent to the pixels. NeoPixels, or WS2812/WS2811, use a single wire protocol that encodes bits as pulses, the time width of the pulses determine whether each bit is a binary 1 or 0. The following is a screenshot from the WS2812 datasheet.

Data transfer time(TH+TL=1.25μs±600ns)

T0H	0 code ,high voltage time	0.35us	±150ns
T1H	1 code ,high voltage time	0.7us	±150ns
T0L	0 code , low voltage time	0.8us	±150ns
T1L	1 code ,low voltage time	0.6us	±150ns
RES	low voltage time	Above 50μs	

Sequence chart:



But Netduino runs .NET Micro Framework, which means two important things: your code is interpreted, and there is a garbage collector.

(note: neither of these are over-all disadvantages! In fact, they exist to make your code safer, and makes coding easier)

Interpreted code means that you write your code in C#, but instead of directly being compiled to ARM assembly code (and subsequently, machine code), it is first compiled to "bytecode", and the processor runs an interpreter to read the bytecode, instead of directly executing machine code. The advantage of this is that the code is safer, you can detect errors like overflows, bad memory access, and check object type info. The disadvantage is that it is much slower than directly executing machine code.

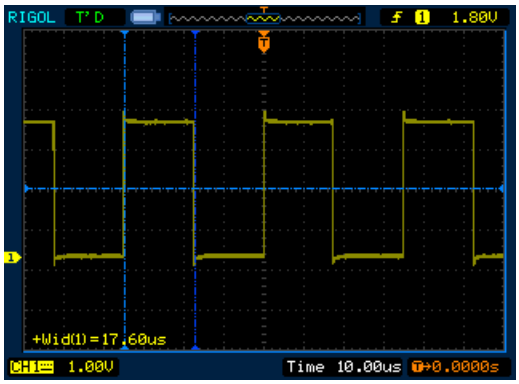
(note: from Wikipedia: "**Managed code** is a term coined by Microsoft to identify computer program source code that requires and will only execute under the management of a **C**ommon **L**anguage **R**untime virtual machine (resulting in bytecode).")

(note: microcontrollers do not normally run interpreted code until you put a interpreter on it, the Netduino is a STM32 microcontroller with a C# CLR interpreter, the BASIC Stamp is a PIC microcontroller with a BASIC interpreter. Arduino is an AVR microcontroller and it does NOT use an interpreter.)

To illustrate my point, I wrote a simple loop in C# that toggles a pin, and observed the time it takes on an oscilloscope:

```
using System;
using System.Threading;
using Microsoft.SPOT;
using Microsoft.SPOT.Hardware;
using SecretLabs.NETMF.Hardware;
using SecretLabs.NETMF.Hardware.Netduino;

namespace Blinky
{
    public class Program
    {
        public static void Main()
        {
            bool pinState = false;

            OutputPort x = new OutputPort(Pins.GPIO_PIN_D0, pinState);

            while (true)
            {
                pinState = !pinState;
                x.Write(pinState);
            }
        }

    }
}
```

Here is the observed signal on the pin. This is a Netduino Plus 2, which runs at 168 MHz, the other Netduino variations all run at different speed.



It takes 17.6 microseconds to toggle that pin. Now go back to the WS2812 datasheet, and you'll realize that it is impossible to meet the timing requirements using Netduino because the signals needs to be under 1 microsecond.

The other problem is the garbage collector, which runs "once in a while". The garbage collector basically prevents memory leaks when you don't explicitly free memory you don't need anymore. But if it runs while we are pulsing signals to the WS2812, it might extend a pulse and violate the timing requirements. This problem can be solved by making sure that interrupts are disabled when we communicate with the WS2812.

Fortunately, .NET Micro Framework and the Netduino firmware are open source. This means it is possible to write native code functions into the firmware, and call these functions from the managed code.

(note: calling functions between managed and unmanaged code is referred to as **interop**, you might have come across this term if you tried to use DLLs in C# before)

Our goal is to write a function in C or C++ that will toggle a pin with strict uninterruptable timing, in order to communicate to WS2812 NeoPixels. This function will then be compiled as a part of Netduino's firmware, and we will update the Netduino with this new modified firmware. After that, we should be able to call our new function from within the C# code.

Building Firmware

This step is a prerequisite step, just to check if you are able to build anything at all, solve any problems early on before you get too far ahead. You might have done this before, so skip it if you want.

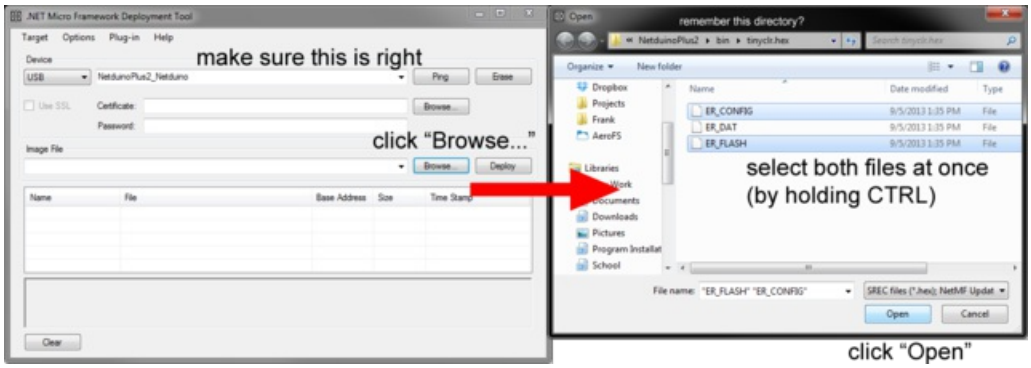
You should have already be able to use your Netduino already. Which means you have already downloaded and installed everything from http://netduino.com/downloads/ (https://adafru.it/cF6) . The most important (for this tutorial) tool inside is MFDeploy.exe, which is found in "Microsoft .NET Micro Framework\v4.2\Tools". (alternatively, you can get MFDeploy from http://www.microsoft.com/en-ca/download/details.aspx?id=5475 (https://adafru.it/cF7) directly)

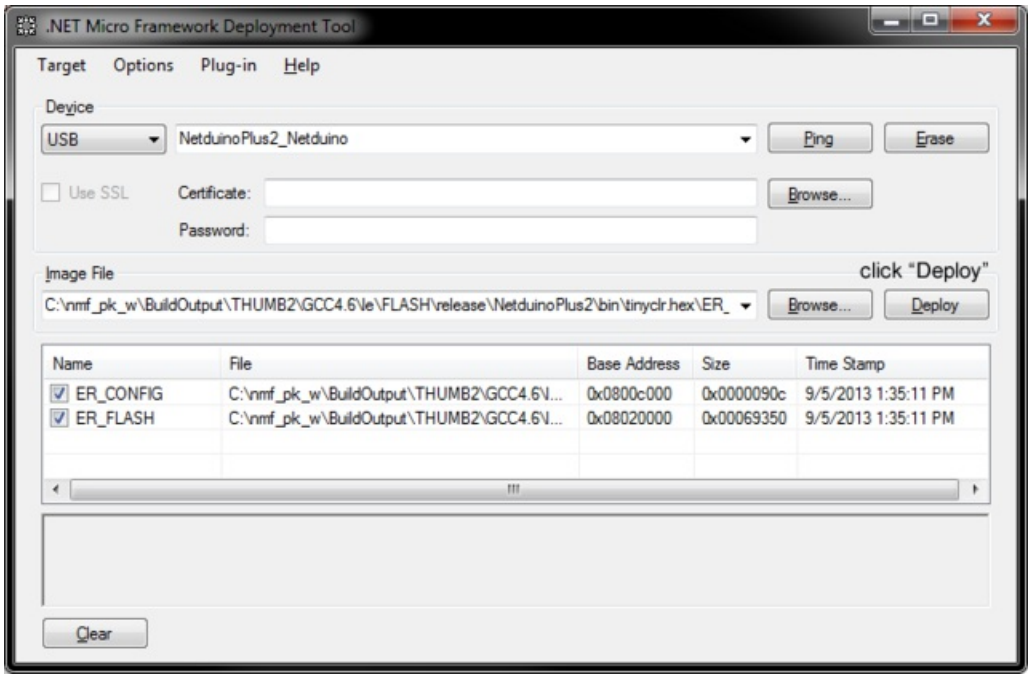
Then you need to follow this guide first: http://wiki.netduino.com/Building-Netduino-Plus-2-Firmware-4-2-2-0-with-Yagarto-GCC-4-6.ashx?HL=gcc (https://adafru.it/cF8) , but after the build finishes with no errors, you do not need to generate the DFU file.

(note: this is extremely time consuming, the downloads are huge, the installations are huge, the compile time takes 5 minutes on a high end computer, and the error messages might be confusing to the beginner. I don't mind if you skip these steps and just flash my precompiled firmware instead)

If you were able to complete this, there will be a folder that looks like **portingkit\BuildOutput\THUMB2\GCC4.6\le\FLASH\release\NetduinoPlus2\bin\tinyclr.hex**, and inside there will be some files (we need **ER_CONFIG** and **ER_FLASH**). Plug in your Netduino, and run **MFDeploy.exe** . See the screenshots below for the next steps.

> I've provided the firmware image files ER_CONFIG and ER_FLASH, plus all the DLL files you need. They are all in the download page. But the DLLs will not work unless you use MFDeploy to update the firmware first.





After clicking "Deploy", the firmware on your Netduino will be updated. Remember how to do this, because you'll be doing this a lot if you are constantly testing custom firmware.

Preparing Custom Firmware

(note: these steps are loosely based on instructions from http://adeneo-embedded.blogspot.ca/2009/01/using-interop-in-net-micro-framework.html (https://adafru.it/cF9))

You may skip this step if you simply want to download the pre-compiled firmware files I've provided in the download page.

The next steps are to create the method signature for your function. We need the signature (think of function prototypes) of the method so that the C# code can call on the native function. I recommend that this is done inside a public static class as a static method. Do this in an entire separate project (use the project template for a .NET Micro Framework static library). My project is named "NeoPixelNative" and the only file inside is "NeoPixelNative.cs", see code below:

```
using System;
using System.Runtime.InteropServices;
using System.Runtime.CompilerServices;

namespace NeoPixel
{
    public static class NeoPixelNative
    {
        /// <summary>
        /// This is the interop call to the low level native C++ code that resides in the modified firmwa
        /// The firmware must contain the NeoPixel low level native C++ code
        /// This method is "internal" so that NeoPixelChain may access it
        /// </summary>
        /// <param name="dataPtr">array of bytes already organized in the GRB format, ready to be sent to
        /// <param name="count">the number of NeoPixels</param>
        /// <param name="pin">The Cpu.Pin representation of which MCU pin the first NeoPixel's data input
        [MethodImpl(MethodImplOptions.InternalCall)]
        public extern static void Write(byte[] dataPtr, int count, UInt32 pin);
    }
}
```

The line "[MethodImpl(MethodImplOptions.InternalCall)]" is an attribute that tells the compiler that the function is internal to the firmware.

(note: only a few data types are supported for the parameters, don't go too crazy, try to pass in only basic data types)

You also need to go into the project properties, look for the ".NET Micro Framework" tab, and check the "Generate native stubs for internal methods" checkbox. Give it the stub a root name (I used "NeoPixel" as the root name) and remember the path (which should be inside your project directory).

Once all these steps are done, build the project. The stubs should be generated. The files you should see are:

- dotNetMF.proj
- NeoPixel_NeoPixel_NeoPixelNative.h
- NeoPixel.cpp
- NeoPixel_NeoPixel_NeoPixelNative_mshl.cpp
- NeoPixel.h
- NeoPixelNative.featureproj
- NeoPixel_NeoPixel_NeoPixelNative.cpp

Remember that "dotNetMF.proj" keeps track of which files are compiled by the C or C++ compiler, while "NeoPixelNative.featureproj" creates the relationship between native code and C#.

(note: this means that for other projects, if at anytime you want to add in another C or C++ code file, edit the "dotNetMF.proj" file)

Edit the "NeoPixelNative.featureproj" file to change the paths involved. See below (the commented XML tags are what they used to be)

```
<?xml version="1.0" encoding="utf-8"?>
<Project ToolsVersion="4.0" xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <PropertyGroup>
    <FeatureName>NeoPixelNative</FeatureName>
    <Guid>{26692d6f-6cac-45db-b560-1904d6a64bf9}</Guid>
    <Description><Add Feature Description Here></Description>
    <Groups ></Groups>
  </PropertyGroup>
  <ItemGroup>
    <DriverLibs Include="NeoPixelNative.$(LIB_EXT)" ></DriverLibs>
    <InteropFeature Include="NeoPixelNative" ></InteropFeature>
    <!-- <MMP_DAT_CreateDatabase Include="$(BUILD_TREE_CLIENT)\pe\$(ENDIANNESS)\NeoPixelNative.pe" ></MMP
    <MMP_DAT_CreateDatabase Include="$(SPOCLIENT)\Solutions\NetduinoPlus2\ManagedCode\NeoPixelNative\bin\
    <!-- <RequiredProjects Include="C:\Projects\netduinoplus2\projects\NeoPixel\NeoPixelNative\Stubs\dotn
    <RequiredProjects Include="$(SPOCLIENT)\Solutions\NetduinoPlus2\DeviceCode\NeoPixelNative\dotNetMF.pr
  </ItemGroup>
</Project>
```

I also added two files to the "dotNetMF.proj" file, I added "NeoPixel_NativeCode.cpp" and "NeoPixel_NativeCode.h", see below:

```
<Project DefaultTargets="Build" ToolsVersion="3.5" xmlns="http://schemas.microsoft.com/developer/msbuild/
  <PropertyGroup>
    <Directory>DeviceCode\Targets\Native\Interop\NeoPixelNative</Directory>
    <AssemblyName>NeoPixelNative</AssemblyName>
  </PropertyGroup>
  <Import Project="$(SPOCLIENT)\tools\targets\Microsoft.SPOT.System.Settings" ></Import>
  <PropertyGroup>
    <OutputType>Library</OutputType>
  </PropertyGroup>
<ItemGroup>
<HFile Include="NeoPixel.h" ></HFile>
<Compile Include="NeoPixel.cpp" ></Compile>
<Compile Include="NeoPixel_NeoPixel_NeoPixelNative.cpp" ></Compile>
<HFile Include="NeoPixel_NeoPixel_NeoPixelNative.h" ></HFile>
<Compile Include="NeoPixel_NeoPixel_NeoPixelNative_mshl.cpp" ></Compile>
<HFile Include="NeoPixel_NeoPixel_NeoPixelNative.h" ></HFile>
<Compile Include="NeoPixel_NativeCode.cpp" ></Compile>
<HFile Include="NeoPixel_NativeCode.h" ></HFile>
</ItemGroup>
<Import Project="$(SPOCLIENT)\tools\targets\Microsoft.SPOT.System.Targets" ></Import>
</Project>
```

I have already created and fully written the new CPP and H files inside the same directory.

Copy the stubs over to "$(SPOCLIENT)\Solutions\NetduinoPlus2\DeviceCode\NeoPixelNative\". "$(SPOCLIENT)" represents where you put your porting kit.

Now look inside your project folder again, look for a folder "projectfolder\NeoPixelNative\bin\Debug\le\" and inside there will be a file "NeoPixelNative.pe". Just make sure it's there. Copy the entire "projectfolder\NeoPixelNative\bin\" into "$(SPOCLIENT)\Solutions\NetduinoPlus2\ManagedCode\NeoPixelNative\bin\"

Then you need to find ".....\Solutions\NetduinoPlus2\TinyCLR\TinyCLR.proj", edit it. Add the line

```
<Import Project="$(SPOCLIENT)\Solutions\NetduinoPlus2\DeviceCode\NeoPixelNative\NeoPixelNative.featurepro
```

before

```
<Import Project="$(SPOCLIENT)\tools\targets\Microsoft.SPOT.System.Interop.Settings" />
```

and then add

```
  <ItemGroup>
    <RequiredProjects Include="$(SPOCLIENT)\Solutions\NetduinoPlus2\DeviceCode\NeoPixelNative\dotNetMF.pr
    <DriverLibs Include="NeoPixelNative.$(LIB_EXT)" ></DriverLibs>
  </ItemGroup>
```

before

```
<Import Project="$(SPOCLIENT)\tools\targets\Microsoft.SPOT.System.Targets" />
```

Bonus: enable bitmaps and other graphical features

Netduino's firmware does not actually support graphical features, if you tried to even create a bitmap object, it will throw a nasty "NotSupportedException".

The .NET Micro Framework does have some pretty cool graphical features but the Netduino does not include them in order to save some space. I've modified our "dotNetMF.proj" file to include some of these features, by adding

```
<Import Condition="'$(PLATFORM_EMULATED_FLOATINGPOINT)'=='true'" Project="$(SPOCLIENT)\Framework\Featur
<Import Project="$(SPOCLIENT)\Framework\Features\Graphics_PAL.libcatproj" />
<Import Project="$(SPOCLIENT)\Framework\Features\Graphics_BMP_CLR.libcatproj" />
<Import Project="$(SPOCLIENT)\Framework\Features\Graphics_Gif_CLR.libcatproj" />
<Import Project="$(SPOCLIENT)\Framework\Features\Graphics_JPG_CLR.libcatproj" />
<Import Project="$(SPOCLIENT)\Framework\Features\Graphics_CLR.libcatproj" />
<Import Project="$(SPOCLIENT)\Framework\Features\SPOT_Graphics_CLR.libcatproj" />
<ItemGroup>
  <PlatformIndependentLibs Include="SPOT_Graphics.$(LIB_EXT)" />
  <RequiredProjects Include="$(SPOCLIENT)\CLR\Libraries\SPOT_Graphics\dotnetmf.proj" />
</ItemGroup>
<ItemGroup>
  <PlatformIndependentLibs Include="Graphics.$(LIB_EXT)" />
  <RequiredProjects Include="$(SPOCLIENT)\CLR\Graphics\dotnetmf.proj" />
</ItemGroup>
<ItemGroup>
  <PlatformIndependentLibs Include="Graphics_BMP.$(LIB_EXT)" />
  <RequiredProjects Include="$(SPOCLIENT)\CLR\Graphics\Bmp\dotnetmf.proj" />
</ItemGroup>
<ItemGroup>
  <PlatformIndependentLibs Include="Graphics_JPEG.$(LIB_EXT)" />
  <RequiredProjects Include="$(SPOCLIENT)\CLR\Graphics\Jpeg\dotnetmf.proj" />
</ItemGroup>
<ItemGroup>
  <PlatformIndependentLibs Include="Graphics_GIF.$(LIB_EXT)" />
  <RequiredProjects Include="$(SPOCLIENT)\CLR\Graphics\Gif\dotnetmf.proj" />
</ItemGroup>
<ItemGroup>
  <DriverLibs Include="Graphics_pal.$(LIB_EXT)" />
  <RequiredProjects Include="$(SPOCLIENT)\DeviceCode\pal\graphics\dotNetMF.proj" />
</ItemGroup>
```

There are plenty of other features within .NET Micro Framework that is not included in the Netduino firmware. I hope this gives you a clue on how to include new features.

Although, you are supposed to add those items to "TinyCLR.proj" instead, but I added them to NeoPixelNative's own "dotNetMF.proj" because I wanted to make my edits to the firmware less intrusive.

Writing the Native Function

You may skip this step if you simply want to download the pre-compiled firmware files I've provided in the download page.

In the previous step, I said I added "NeoPixel_NativeCode.cpp" and "NeoPixel_NativeCode.h". We now have to write these files. The CPP file will contain the actual function and private definitions while the H file will contain the public function pointer and public definitions.

I avoided writing too many things into the auto-generated stub files, but I did make the stub function call my own function.

The function to send data to the NeoPixels is very simple.

1. The first thing it does is translate Netduino's pin assignment from an integer to the specific hardware output port and bit. (this was sort of confusing to do, but I got some clues after reading the file "STM32_GPIO_functions.cpp")
2. Then it disables all interrupts so it can output pulses with correct uninterrupted timing.
3. Then it iterates through all the bits in all the bytes, inserting the correct (read the WS2813 datasheet) amount of delay using delay loops (a for-loop macro made with an volatile down-counting integer and a NOP instruction)
4. Finally, it re-enables all interrupts

```
#include <TinyCLR_Interop.h>

#if defined(PLATFORM_ARM_STM32F4_ANY)
#include <DeviceCode/stm32f4xx.h>
#elif defined(PLATFORM_ARM_STM32F2_ANY)
#include <DeviceCode/stm32f2xx.h>
#else
#include <DeviceCode/stm32f10x.h>
#endif

#include "NeoPixel.h"
#include "NeoPixel_NeoPixel_NeoPixelNative.h"
#include "NeoPixel_NativeCode.h"

#define NP_DELAY_LOOP(x) do { volatile int ___i = (x); while (___i--) { asm volatile ("nop"); } } while (

// these numbers were tuned specifically for Netduino Plus 2, STM32F4 running at 168MHz
// tuning done using an o'scope
#define NP_WAIT_T1H() NP_DELAY_LOOP(12) // 700 ns needed, tested 12 = 680ns
#define NP_WAIT_T1L() NP_DELAY_LOOP(10) // 600 ns needed, tested 10 = 580ns
#define NP_WAIT_T0H() NP_DELAY_LOOP(5)  // 350 ns needed, tested 5  = 350ns
#define NP_WAIT_T0L() NP_DELAY_LOOP(15) // 800 ns needed, tested 15 = 800ns

void NeoPixelNativeWrite(CLR_RT_TypedArray_UINT8 data, INT32 count, UINT32 pin)
{
 GPIO_TypeDef* port = ((GPIO_TypeDef *) (GPIOA_BASE + ((pin & 0xF0) << 6)));
 UINT16 *_BSRRH, *_BSRRL;
 // note: I think the BSRRx registers are reversed in the declaration, hence why this code seems reversed
 #if defined(PLATFORM_ARM_STM32F2_ANY) || defined(PLATFORM_ARM_STM32F4_ANY)
 _BSRRH = (UINT16*)&(port->BSRRL); _BSRRL = (UINT16*)&(port->BSRRH);
 #else
 _BSRRH = (UINT16*)&(((UINT16*)(&port->BSRR))[1]);  _BSRRL = (UINT16*)&(((UINT16*)(&port->BSRR))[0]);
 #endif
 UINT8 portBit = 1 << (pin & 0x0F);
 *_BSRRL = portBit; // clears the pin low
```

```
  __disable_irq(); // disable interrupts so that timing is accurate

  // pin is already assumed low
  // data is already in GRB order
  INT32 byteIdx, bitMask;
  INT32 cx3 = count * 3; // 3 bytes per neopixel
  for (byteIdx = 0; byteIdx < cx3; byteIdx++) // for every byte in the array
  {
   for (bitMask = 0x80; bitMask > 0; bitMask >>= 1) // MSB first
   {
    *_BSRRH = portBit; // sets the pin high
    if ((data[byteIdx] & bitMask) != 0)
    {
     NP_WAIT_T1H();
     *_BSRRL = portBit; // clears the pin low
     NP_WAIT_T1L();
    }
    else
    {
     NP_WAIT_T0H();
     *_BSRRL = portBit; // clears the pin low
     NP_WAIT_T0L();
    }
   }
  }

  __enable_irq();
 }
```

Please download the files to see them all (all project downloads are in another page, see the navigation panel)

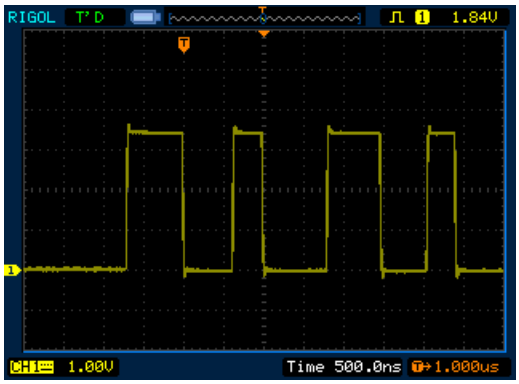
If everything goes right, you should be able to build and update the firmware, by going through the same steps I've already shown you.

Testing and Verification

OK! I've done the grunt work for you (It took several trial-and-error tests to get right!) While working on it, I noticed these things:

- the "BSRRL" and "BSRRH" registers are apparently role-reversed in the file that defines hardware peripheral registers. I had to swap them inside my code.
- the overhead of the CLR is so long that I didn't need to add the 50 microsecond delay required to latch all the NeoPixels

I also had to tune the delay loop counts manually, which is common when dealing with these delicate bit-bang encoding. However, they should work great now - see o'scope screenshot below (it shows binary 1 0 1 0)



Some tips for when you write your own firmware:

Look around the porting kit and Netduino firmware for files that might contain helpful functions and definitions. For example, some timing and delay utilities are found inside "C:\PortingKitPath\DeviceCode\Targets\Native\STM32\DeviceCode\STM32_Time\STM32_time_functions.cpp". STM32F4's hardware definitions are inside "C:\PortingKitPath\DeviceCode\Targets\Native\STM32\DeviceCode\system_stm32f4xx.h", and I'm sure there are other useful files.

But you need to know how to include these files, because the paths are so messed up. Here's a list of paths that are included by the compiler (via the "-I" option)

- c:\PortingKitPath\DeviceCode\pal\Double
- c:\PortingKitPath\DeviceCode\include
- c:\PortingKitPath\DeviceCode\Cores\arm
- c:\PortingKitPath\Support\Include
- c:\PortingKitPath\crypto\inc
- c:\PortingKitPath\CLR\Include
- c:\PortingKitPath\CLR\Libraries\CorLib
- c:\PortingKitPath\CLR\Libraries\SPOT
- c:\PortingKitPath\CLR\Libraries\SPOT_Hardware
- c:\PortingKitPath\CLR\Libraries\SPOT_Graphics
- c:\PortingKitPath\CLR\Libraries\SPOT_Net
- c:\PortingKitPath\Solutions\NetduinoPlus2
- c:\PortingKitPath\devicecode\Targets\Native\STM32
- c:\PortingKitPath\DeviceCode\Cores\arm
- c:\PortingKitPath\DeviceCode\Cores\arm\Include
- c:\PortingKitPath\DeviceCode
- c:\PortingKitPath\DeviceCode\Include
- c:\PortingKitPath\CLR\Include
- c:\PortingKitPath\DeviceCode\Targets\Native\STM32\NetduinoPlus2
- c:\PortingKitPath\Support\Include
- c:\PortingKitPath\DeviceCode\include
- c:\PortingKitPath\CLR\include
- c:\PortingKitPath\Crypto\inc
- c:\PortingKitPath\DeviceCode\include

Using " **#include** <> "relative to any of these paths will allow your file to be found.

Be prepared to open "msbuild.log" a lot! I suggest keeping it open in Notepad++ because every time it changes,

Notepad++ will ask you if you want to reload the file, which is faster than reopening it manually.

This build log file is 3 MB in size most of the time! Have fun looking for the errors inside it. The very last part of the log indicates which step it failed, but not why. But if it is an error involving the compiler (gcc) or the linker (ld), then search the build log for

```
: error:
```

and you should be able to find what exactly went wrong.

Each full build or rebuild takes 5 minutes, each partial rebuild takes around 1 minute. This is on my computer with a SATA 3 SSD and Intel Core i7 Ivy Bridge. If you are doing this often, get a snack, turn on a TV...

Writing a Good C# Class

You may skip this step if you simply want to download the pre-compiled DLL files (NeoPixel.dll and NeoPixelNative.dll) from the download page. The DLL files will not work until the firmware is updated as well (using MFDeploy to load ER_CONFIG and ER_FLASH), which is covered in the previous Building Firmware page.

So far our custom function takes raw data and sends the data in sequence to the NeoPixels. It gets the job done with a minimum amount of bells and whistles.

But I thought it would be nice if we can represent each NeoPixel as a class object named NeoPixel, with three values that represent the R G and B color intensities. I used overloads so you can create an instance of NeoPixel using many data types.

I then created another class that organizes a chain of NeoPixels, named it "NeoPixelChain". It inherits from a generic list interface, so you can access it as it was an array of NeoPixels. You can then call "Write" on the entire chain, and the byte sequence will be automatically rearranged into the GRB sequence that the WS2812 expects, then sent out of the output pin using our native function.

(note: this is just to illustrate "inheritance", which is a feature of object oriented programming languages such as C#, Java, C++, and many more)

I also wrote a few more wrapper functions under the class name "NeoPixelWriter". The most impressive one can directly draw a bitmap image directly onto a grid of NeoPixels (such as the NeoPixel 8x8)

(note: NeoPixelWriter makes heavy use of "overloading")
(note: although "NeoPixelWriter" has the word "Writer" in it, it is does not inherit from "Stream" or even resemble "Stream" at all, sorry about that)

Remember, on the Netduino, you can easily load a bitmap from a SD card or download from the network, or draw one during runtime using graphic functions. Have fun with this one!

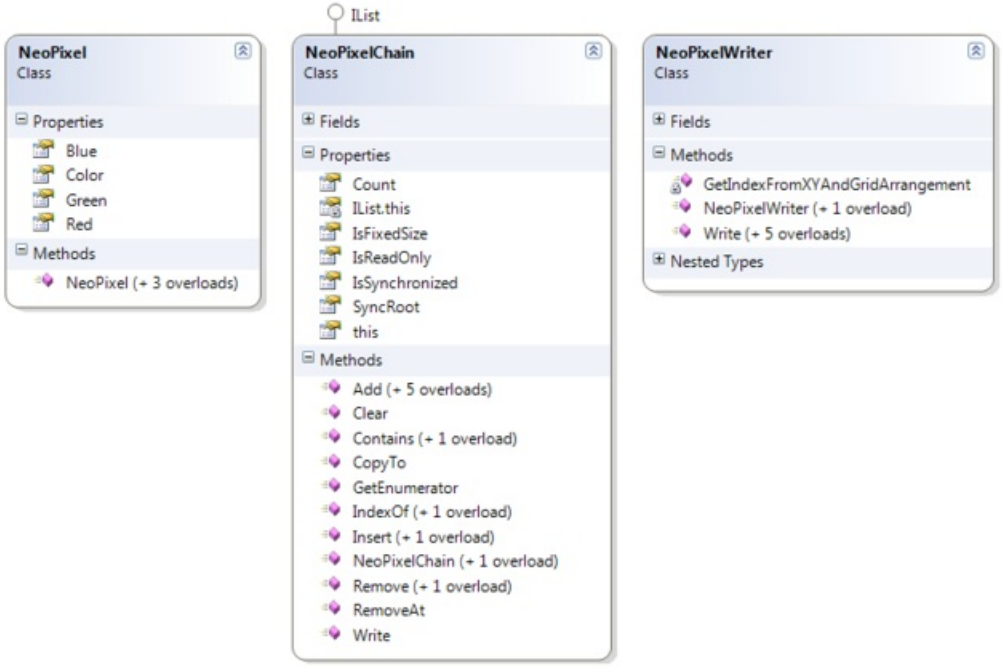
All the files are provided on the download page, see the navigation panel.

All documentation about the API is done within the code, which means Visual Studio can automatically show you the documentation when:

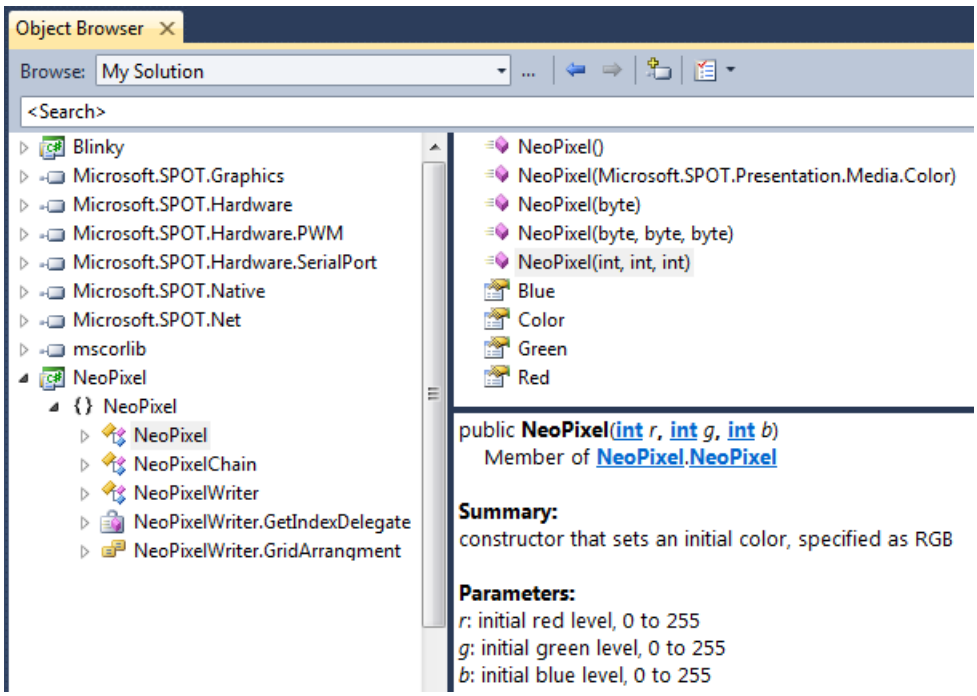
- you use "Intellisense" (aka autocomplete)
- use "Object Browser"

But in order to do this, you need both the DLL and documentation XML files (also provided in downloads) together.

Below is a class diagram of my "NeoPixel" namespace.



Here is "Object Browser" in action, showing you how to view the code documentation

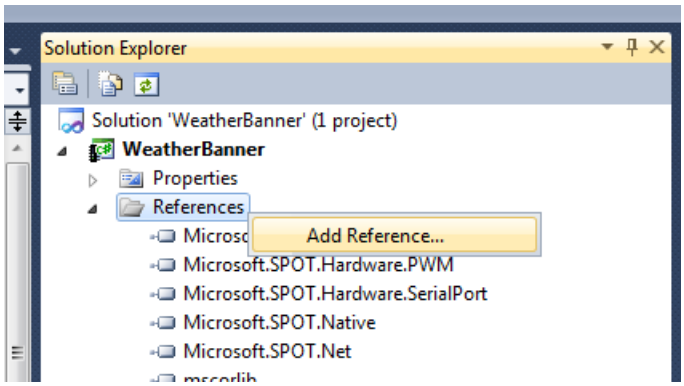


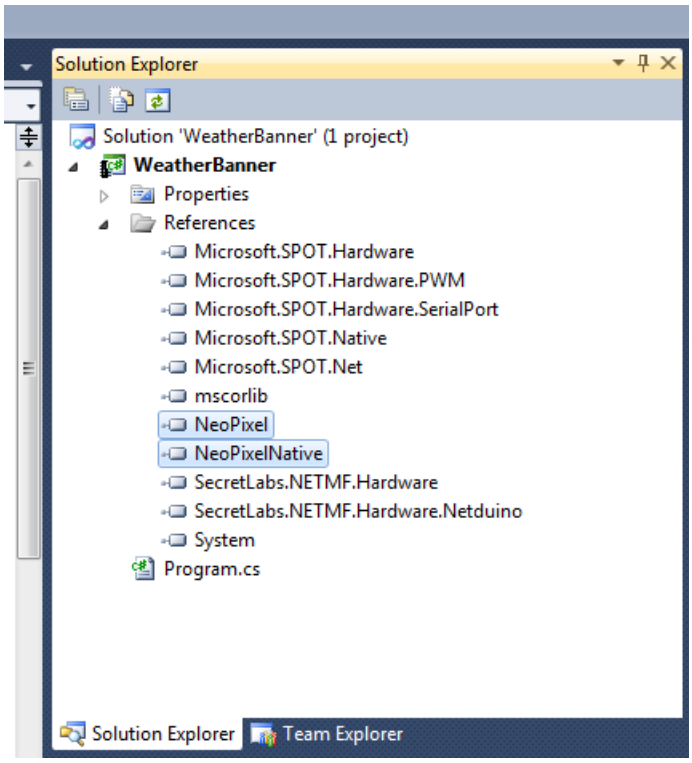
Using the C# Class

Now it's time to add NeoPixel functionality into your own projects.

You should already have two files, **NeoPixel.dll** and **NeoPixelNative.dll**. You've either downloaded them or compiled them yourself.

In the project where you want to use NeoPixels, inside Visual Studio, you need to add references to these two DLL files.





You actually need all the files from the bin folder of NeoPixel (not just the DLLs). Alternatively, just include the entire NeoPixel and NeoPixelNative projects into your solution and include them that way so the files are automatically found.

And in the code, you might want to add the following line where you have all your "using"

```
using NeoPixel;
```

For example (this program will give you a nice colorful test pattern)

```
using System;
using System.Threading;
using Microsoft.SPOT;
using Microsoft.SPOT.Hardware;
using SecretLabs.NETMF.Hardware;
using SecretLabs.NETMF.Hardware.Netduino;

using NeoPixel;

namespace NeoPixelTester
{
    public class Program
    {
        public static void Main()
        {
            NeoPixelChain chain = new NeoPixelChain(Pins.GPIO_PIN_D0);

            for (int i = 0; i < 8; i++)
            {
                int v = (i + 0) * (256 / 80) + 1;
                chain.Add(0x55, 0xAA, 0x5A);
                chain.Add(v, v, v);
                chain.Add(0, 0, v);
                chain.Add(0, v, 0);
                chain.Add(0, v, v);
                chain.Add(v, 0, 0);
                chain.Add(v, 0, v);
                chain.Add(v, v, 0);
            }

            chain.Write();

            while (true)
            {
                // do nothing
            }
        }
    }
}
```

To enable advanced graphical features such as bitmaps and fonts, you also need to add reference to **Microsoft.SPOT.Graphics**

Download

All source code and compiled binary files are available from my own github:
https://github.com/frank26080115/NeoPixel-on-NetduinoPlus2/ (https://adafru.it/cFj)

There is an option to download everything as a ZIP file.

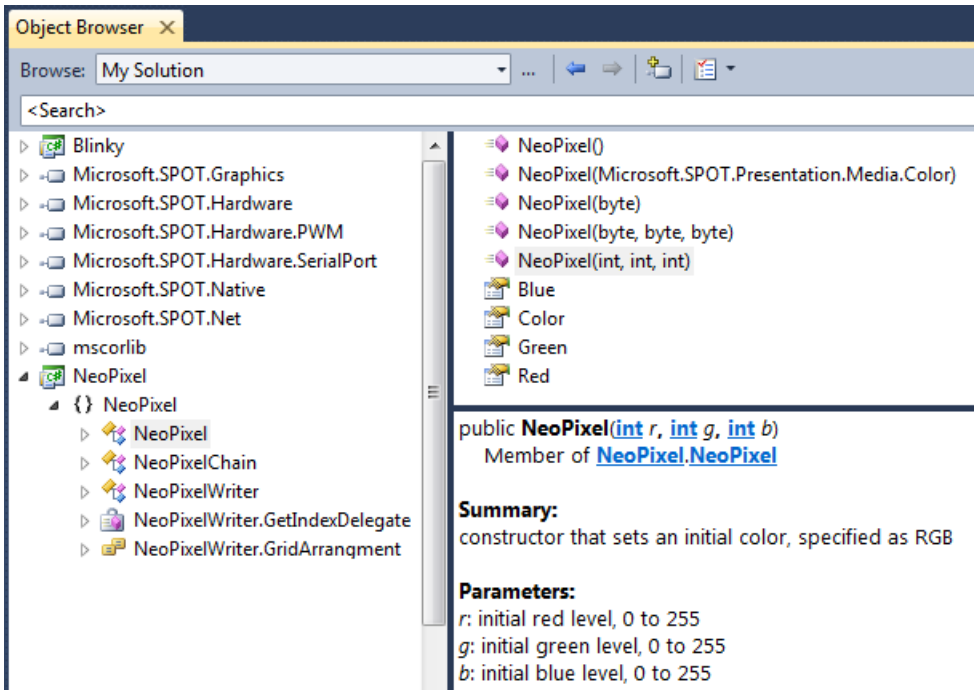
The directory structure is organized such that you can overwrite the porting kit directory if you've already placed Netduino Plus 2 into it (see the step called **Building Firmware** if this confuses you)

If you are only interested in pre-compiled binary and DLL files, you are looking for these specific files:

- https://github.com/frank26080115/NeoPixel-on-NetduinoPlus2/...../ER_FLASH (https://adafru.it/BoC)
- https://github.com/frank26080115/NeoPixel-on-NetduinoPlus2/...../ER_CONFIG (https://adafru.it/BoD)
- All files inside https://github.com/frank26080115/NeoPixel-on-NetduinoPlus2/...../bin/..... (https://adafru.it/BoE)
- All files inside https://github.com/frank26080115/NeoPixel-on-NetduinoPlus2/...../bin/..... (https://adafru.it/BoF)

> Everything I have done only works on Netduino Plus 2, using its V4.2.2.0 firmware. It does not work with any other models of Netduino. When firmware V4.3 becomes available, I will do a quick check to see if everything still works.

All code usage documentation are provided within the corresponding documentation XML files accompanying the DLL files. If these files are kept together, you can use the "Object Browser" or "Intellisense" (aka autocomplete) to view the documentation.



Bonus: Bitmap + Text

Here's a cool demo you can try to make. I'll be using a 8x8 NeoPixel panel. The code shall draw a background image on the panel, and then overlay some text on it as well.

First, I created a bitmap file, 8 by 8 pixels. It looks like the following: (well... it is a really REALLY small file)

Here's a zoomed in version of the same file

I saved it as "examplebitmap.bmp".

In your C# project, use "add a new file" to add "MyResources.resx". Open "MyResources.resx" and use "add existing file" to add "examplebitmap.bmp". This is the fastest way to embed an image into your project.

(note: this takes up a lot of your precious flash memory! remember: the Netduino Plus 2 also has a microSD card slot you can use, or you can even download images from the network since it has built-in Ethernet)

(note: I have included JPEG and GIF decoding, but I haven't tried them out yet)

To use the resource, you can simply write

```
Bitmap bm = MyResources.GetBitmap(MyResources.BitmapResources.examplebitmap);
```

> There has been a bug report that Visual Studio 2012 does not process the bitmap resource properly. I know Visual Studio 2010 does work.

Next, we'll embed a font into the project so we can draw text.

We are using a 8 pixel tall panel so it makes sense to use a font specifically designed to be 8 pixels tall. I'm using one called "Apple][" from http://www.dafont.com/apple.font (https://adafru.it/cFk). Download the .ttf file.

Use "Tiny Font Tool" from http://informatix.miloush.net/microframework/Utilities/TinyFontTool.aspx (https://adafru.it/cFl) to generate a .tinyfnt file. I called mine "apple2.tinyfnt"

(note: the porting kit comes with another tool called "TFConvert.exe" to do the same thing but "Tiny Font Tool" is MUCH better, thank you Miloush!)

Open "MyResources.resx" and use "add existing file" to add "apple2.tinyfnt".

You can now use the font like so:

```
Font f = MyResources.GetFont(MyResources.FontResources.apple2);
```

Make sure that in your project references, you've added "Microsoft.SPOT.Graphics"

Now back to our original goal: put some text on a bitmap, and show it on the NeoPixel panel.

```
using System;
using System.Threading;
using Microsoft.SPOT;
using Microsoft.SPOT.Hardware;
using SecretLabs.NETMF.Hardware;
using SecretLabs.NETMF.Hardware.Netduino;

using NeoPixel;
using Microsoft.SPOT.Presentation.Media;

namespace NeoPixelBitmapTextDemo
{
    public class Program
    {
        public static void Main()
        {
            NeoPixelWriter w = new NeoPixelWriter(Pins.GPIO_PIN_D0);

            while (true)
            {
                foreach (char c in "ABCabc")
                {
                    for (int i = -8; i < 8; i++)
                    {
                        Bitmap bm = MyResources.GetBitmap(MyResources.BitmapResources.examplebitmap);
                        Font f = MyResources.GetFont(MyResources.FontResources.apple2);
                        bm.DrawText("" + c, f, Color.White, 1, i);
                        w.Write(bm, NeoPixelWriter.GridArrangment.LeftToRight_TopToBottom_Zigzag);
                        Thread.Sleep(100);
                    }
                }
            }
        }
    }
}
```

(3MB GIF animation above, please wait for it to load)