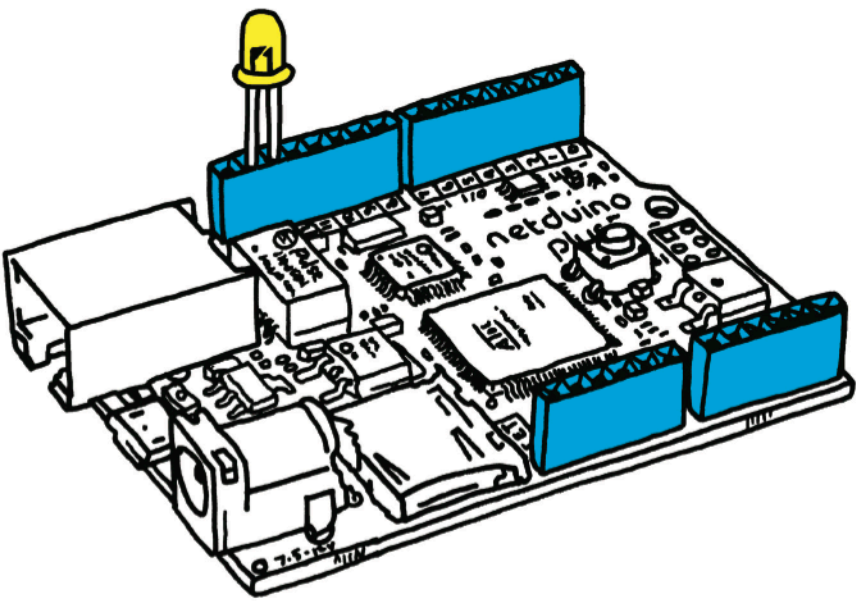
Make: PROJECTS

PROGRAMMING
OPEN SOURCE
ELECTRONICS
WITH .NET

Getting Started with Netduino

Chris Walker



O'REILLY

Make:
makezine.com

O'Reilly Ebooks—Your bookshelf on your devices!

When you buy an ebook through oreilly.com, you get lifetime access to the book, and whenever possible we provide it to you in four, DRM-free file formats—PDF, .epub, Kindle-compatible .mobi, and Android .apk ebook—that you can use on the devices of your choice. Our ebook files are fully searchable and you can cut-and-paste and print them. We also alert you when we've updated the files with corrections and additions.

Learn more at http://oreilly.com/ebooks/

You can also purchase O'Reilly ebooks through iTunes, the Android Marketplace, and Amazon.com.

Getting Started with Netduino

Chris Walker

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

Getting Started with Netduino

by Chris Walker

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Editor: Brian Jepson
Production Editor: Kristen Borg
Proofreader: O'Reilly Production Services
Cover Designer: Karen Montgomery
Interior Designer: Ron Bilodeau and Edie Freedman
Illustrator: Marc de Vinck

February 2012: First Edition.

Revision History for the First Edition:
February 9, 2012 First release

See *http://oreilly.com/catalog/errata.csp?isbn=9781449302450* for release details.

ISBN: 978-1-449-30245-0
[LSI]
1328800561

This book is dedicated to the Netduino Community:

Tens of thousands of tinkerers have picked up Netduino and started tinkering.

So many have shared their experiences, projects, and friendship.

Thank you for making our journey so fun and inspiring.

Contents

Preface

Computers surround us. I'm not speaking of laptops or tablets or cell phones. Billions of remote controls, thermostats, sensors, and gadgets of all sorts have little computers inside them. And while millions of software engineers develop applications for phones, computers, and the Web—the programming languages and skills that apply there are quite a bit different than those needed to develop code for tiny embedded microcontrollers.

Or at least, they were.

In 2004, Microsoft introduced the SPOT Smart Watch. It ran a tiny version of their desktop .NET programming runtime and enabled application developers to write software for its tiny microcontroller using the C# programming language they already knew. Almost a decade later, this runtime is now in its fourth major version, is running on millions of devices around the world, and has grown to enable tinkerers to use traditional software development skills to build their own electronics projects with Netduino.

Like me, you may be a tinkerer. You may like building things or tearing things apart to understand how they work. You may want to build your own web-based coffee machine, Morse code generator, or electronically enhanced Halloween costume.

Or you may be an educator or student who wants to learn how electronics work. Netduino and the .NET Micro Framework enable you do this without drowning in a sea of datasheets, and without needing to understand the intricacies of microcontroller registers at the same time.

Because Netduino is open source, all design files and source code are included. If you desire to become an expert or just need a reference to understand how things work behind the scenes, that is all provided at no charge. Netduino gives you freedom to build, to remix, and to have fun.

And for many, Netduino is about fun: to learn how electronics work, to build cool projects, and to play. For others, Netduino is a serious tool used to develop viable products.

Whether you are interested in Netduino for fun or for profit, there is an online community of tens of thousands of fellow makers at *http://forums.netduino.com.* Come join us in learning, in building, and in sharing our electronics achievements. I look forward to meeting you there.

What You Need to Know

This book is written with the goal of giving non-programmers enough training to successfully build the samples in this book, while providing software engineers the ability to delve into electronics using their sophisticated programming skills.

If you have written scripts for a web page, you have the skills necessary to tackle this book. If you have used a word processor before, you should be able to follow along with the samples. And if you write desktop or web applications in C# for a living, you should enjoy this journey.

How to Use This Book

This book is intended to be read from beginning to end. I start by introducing Netduino and walking you through installation of the free Windows-based development tools. I then introduce electronic components like buttons, LEDs, and speakers—and show you how to use them. Finally, I show you how to connect an Internet-enabled Netduino to the Web.

These examples are all building blocks, giving you tools in your electronics tool chest. And while it is a lot of fun to write code that changes the color of LEDs or plays music on a tiny speaker, the building blocks are ultimately for use in your own projects.

As you build your own projects or reproduce the projects of fellow Netduino community members, you'll inevitably need electronics parts. Stores like RadioShack and Micro Center can provide you with drawers full of components you can use to build projects, and online stores like MakerShed (*http://www.makershed.com*) provide both parts and full kits (with instructions) that can guide you in your tinkering ways.

Enjoy the journey. It's a lot of fun.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

TIP: This icon signifies a tip, suggestion, or general note.

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Getting Started with Netduino* by Chris Walker (O'Reilly). Copyright 2012 Secret Labs LLC, 978-1-4493-0245-0."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

Safari® Books Online

Safari Books Online is an on-demand digital library that lets you easily search over 7,500 technology and creative reference books and videos to find the answers you need quickly.

With a subscription, you can read any page and watch any video from our library online. Read books on your cell phone and mobile devices. Access new titles before they are available for print, and get exclusive access to manuscripts in development and post feedback for the authors. Copy and paste code samples, organize your favorites, download chapters, bookmark key sections, create notes, print out pages, and benefit from tons of other time-saving features.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at:

http://shop.oreilly.com/product/0636920018032.do

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

This Book Was Made Possible By

In the 18 months since the launch of Netduino, so much has happened...and the fun has only started.

Like Netduino itself, this book couldn't have happened without the help of so many others. Here are just a few of the people that made this all possible.

Thank you to:

You—for purchasing this book and joining the growing worldwide community of Netduino tinkerers. You are why Netduino exists.

Brian Jepson, Limor Fried, and Phillip Torrone—who taught me the importance of open source hardware and have been so supportive of our efforts.

Clint Rutkas and Microsoft's Channel 9 Team—who have celebrated many of the community's awesome projects, and who make hacking electronics fun!

Marc de Vinck and the MakerShed team—who were the first to take a chance on Netduino and bring it to a larger audience online.

Stefan Thoolen—for spending dozens and dozens of hours reviewing the book, making recommendations for sample code, creating the netmftoolbox, and more. Also, for being the best moderator ever. You are a good friend.

The Arduino team, especially Massimo Banzi and Tom Igoe—for kickstarting the open hardware movement, for continuing to introduce so many people to electronics, and for warmly inviting me into their world.

Brian Jepson and Marc de Vinck—thank you a second time—you have been so patient during the making of this book, and then you edited and illustrated it so nicely. You're the best.

Colin Miller, Lorenzo Tessiore, and Zach Libby from Microsoft—thank you for giving life to .NET Micro Framework, for open sourcing it, and for continuing to develop the core platform all these years.

David Stetz, Stanislav Simicek, fellow .NET Micro Framework core tech team members, and others who have contributed to the codebase—your contributions are the lifeblood of .NET Micro Framework.

Miguel de Icaza and the Mono team—for making it possible for Mac and Linux users to write code for Netduino too, and for your open source passion.

Cuno Pfister—your book *Getting Started with the Internet of Things* was the catalyst for this book, and all your contributions to the community are greatly appreciated.

Secret Labs staff—for demanding perfection and for being so sneaky and quiet while I finished this book.

The entire O'Reilly team—for believing in Netduino and for making this book possible.

3/First Projects

Microcontroller boards like Netduino live in a realm that bridges software and electronic hardware. The Netduino itself is the bridge, and your Netduino app determines how software and hardware talk to each other. This means that one of the fundamental activities you'll engage in is *signaling*: your app must send signals over digital outputs and receive signals on digital inputs.

These digital signals are represented in your app's code as binary 1s and 0s. Usually, 1s are represented by a higher voltage across a wire (on Netduino, that's 3.3 or 5 volts) and 0s are represented by a low voltage across a wire (on Netduino, 0 volts).

Using these 1s and 0s, you can send high and low voltages to electronic components like LEDs and relays (turning them on and off). You can also receive high and low voltages from components such as pushbuttons: when your app receives a 1, it means the button is pushed.

Both Netduino and Netduino Plus have an onboard LED that can be controlled from code. As a first project, you'll learn how to send digital 1s and 0s by blinking this LED. Later in this chapter, you'll learn how to read the state of Netduino's onboard pushbutton.

Start Visual Studio

Start Visual C# Express 2010 (if you use the full version of Visual Studio 2010, start Visual Studio 2010 instead). The installer (see "Step 1: Install Visual Studio Express" on page 10) created a folder and shortcut for this program in your Start menu under All Programs (Windows Vista or Windows 7) or Programs (Windows XP).

The Visual Studio programming environment will start. Now you're ready to create your first project:

1. Click on the New Project link. If no link is visible, go to the File menu and select New Project.

2. The New Project window should now pop up. Visual Studio displays a set of installed templates. We want to pick Visual C#→Micro Framework from the list on the left. Then pick Netduino Application from the list on the right. Give your project a name such as Blinky (as shown in Figure 3-1), and click OK.

NOTE: If you haven't unpacked your Netduino, do so now. Attach its sticky feet. Grab a Micro USB cable and plug the Netduino into your computer. If you did not get a Micro USB cable with your Netduino, you may be able to borrow one from a cell phone or another device.

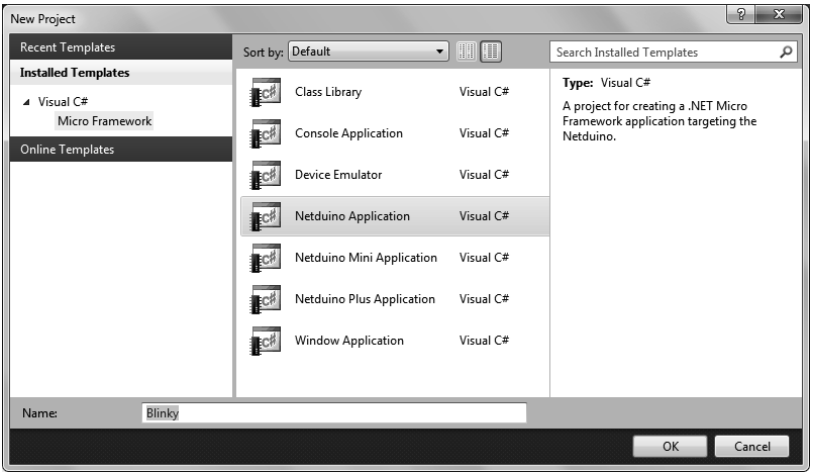


Figure 3-1. *Create a new project*

Blinking the Onboard LED

The first thing that microcontroller programmers often do is blink an LED on their new electronics board. This verifies that the board is booting up properly and that they can successfully create and run a simple app on the board.

While this can often take hours or days in the traditional microcontroller world, you'll do it in a few minutes. Go ahead and write this Blinky app now:

1. Now that Visual Studio is open, look for Solution Explorer on the right side of the screen. Solution Explorer shows the source and data files, which make up your Netduino project. Of particular note, the file *Program.cs* holds the startup code for your project. You're going to open it and write about a half dozen lines of code. Double-click on *Program.cs* now (or right-click on its name and select Open).
2. In the main section of the Visual Studio editor, you are now editing *Program.cs*. Click on the line underneath the text `// write your code here`. This is where you'll write your code.
3. Now, type the following:

```
OutputPort led = new OutputPort(Pins.ONBOARD_LED, false);
```

   This first line of code creates an `OutputPort`. An `OutputPort` is a software object that lets you control the voltage level of a *pin* on the Netduino. The first parameter tells the Netduino which pin of the microcontroller you want to control, and the second parameter tells the Netduino which state to put it in. `Pins.ONBOARD_LED` is shorthand that specifies the Netduino's built-in blue LED. The second parameter (false) puts the LED in an initial state of OFF (false).

NOTE: For digital inputs and outputs, higher voltage (1) is represented as `true` and lower voltage (0) is represented as `false`.

4. Now, you're going to blink the LED on and off repeatedly. A straightforward way to create an action that repeats forever is to put it inside a loop that never ends. Add the following code to your project:

```
while (true)
{
}
```

   The keyword *while* tells the microcontroller to do something in a loop while a certain condition is met. This condition is placed in parentheses. In our case, we use a condition of `true`. Since conditions are met when they are true, putting `true` here means that the loop will repeat forever.
5. Next, create the blinking LED code. Between the two sets of curly braces, insert the following four lines of code:

```
led.Write(true); // turn on the LED
Thread.Sleep(250); // sleep for 250ms
led.Write(false); // turn off the LED
Thread.Sleep(250); // sleep for 250ms
```

Your final program's `Main()` method should now look like the following listing. Figure 3-2 shows Visual Studio Express with the Solution Explorer to the right and the complete *Program.cs* open in the editor:

```
public static void Main()
{
    // write your code here
    OutputPort led = new OutputPort(Pins.ONBOARD_LED, false);

    while (true)
    {
        led.Write(true); // turn on the LED
        Thread.Sleep(250); // sleep for 250ms
        led.Write(false); // turn off the LED
        Thread.Sleep(250); // sleep for 250ms
    }
}
```

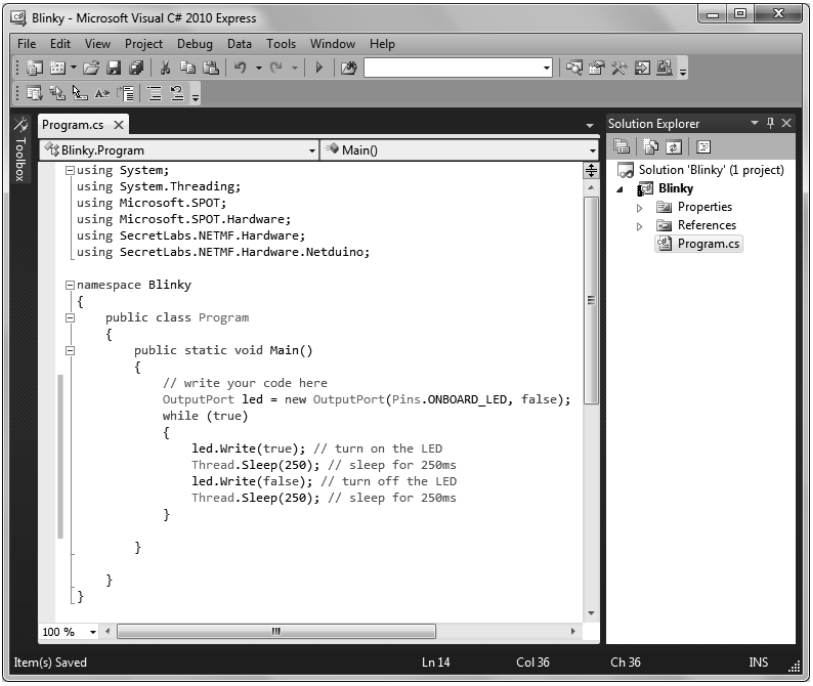


Figure 3-2. *The finished Blinky program*

Running the Blinky App

Next, you'll deploy the Netduino app to the Netduino and watch it run.

By default, Visual Studio runs projects in an *emulator*. This allows software developers to create and test programming logic for a new hardware product before the actual hardware is built. You won't use the emulator since you have a real Netduino, so you must let Visual Studio know that you have physical hardware it should use instead.

Click on the Project menu and select your project's properties (generally, the last item in the Projects menu, such as Blinky Properties). Next, do the following:

1. When the project properties appear, click on the .NET Micro Framework category on the left side.
2. Now you're ready to change the deployment target from the Emulator to the Netduino. Change the Transport from Emulator to USB and then make sure that the Device selection box shows your Netduino, as shown in Figure 3-3. If it doesn't, unplug and reattach your Netduino.

NOTE: If you're using a Netduino Plus, the target name will be different. So if you need to switch between deploying to a Netduino and Netduino Plus, you'll need to return to the project properties and change the Device setting to the device you want to deploy to.

Now, you're ready to run the project. When you run the project, your code is deployed to the Netduino and then automatically started. To run your project, click the Start Debugging button in the toolbar at the top of the screen. It looks like the Play button on a music player. You could also press its keyboard shortcut, F5.

NOTE: You'll just watch the program run for now, but when you start building sophisticated Netduino Apps, you may want to pause or step through your code line-by-line, peek at the values of data in the Netduino's memory, and so forth. If you want to go through the code line-by-line, one way to do that is to start the program with Step Into (F11) instead of Start Debugging.

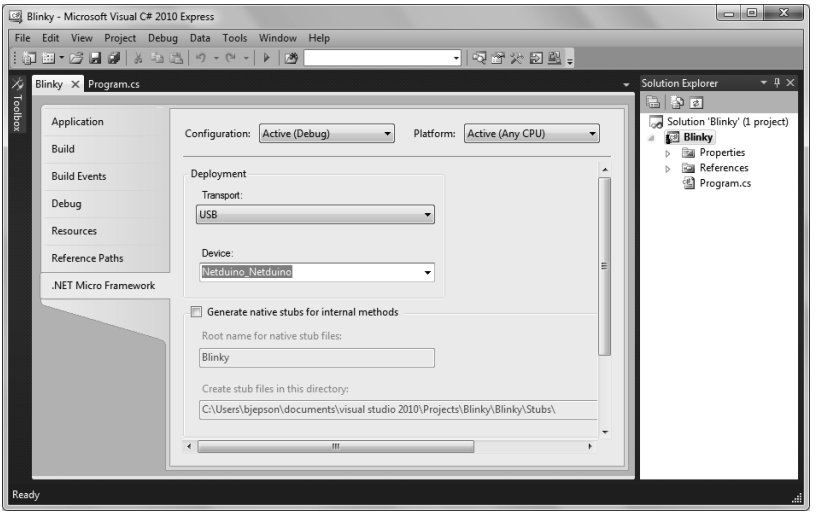


Figure 3-3. *Setting the project's .NET Micro Framework properties*

Visual Studio will now deploy your first Netduino app to the Netduino hardware. In a few seconds, you'll see the blue LED (Figure 3-4) blinking on and off every half second.

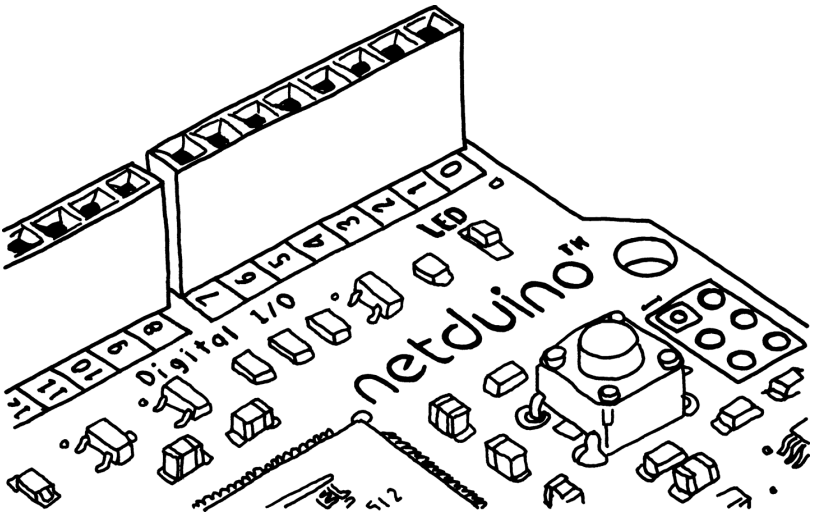


Figure 3-4. *The Netduino's onboard LED*

NOTE: When you started your Netduino app, it was written into the Netduino's microcontroller chip's flash memory, so all you have to do to run the program again is plug it in via a MicroUSB cable or with a power adapter (using the power barrel jack). You can write over your Netduino App. Visual Studio will automatically stop and erase your current Netduino App whenever deploying a new one.

Now that you've successfully written a simple Netduino app and controlled an LED by sending it digital output of high (true) and low (false) voltages, you're ready to learn how to read digital input signals.

Pushing the Onboard Button

Both Netduino and Netduino Plus have an on onboard pushbutton labeled SW1. You can read the digital value of this pushbutton—true or false—to determine whether it is currently being pushed. Then you can take actions based on that pushbutton's current state, such as turning an LED on and off or instructing a Netduino-powered robot to start moving its motors.

NOTE: By default, this pushbutton will reboot your Netduino and restart your current Netduino app. But because of a special configuration in the Netduino's circuitry, it can alternatively be used as a digital input.

In Visual Studio, create a new project by selecting New Project in the File menu. Select Netduino Application as the template as before, and give your project a name such as "Pushbutton1." Finally, double-click on *Program.cs* in the Solution Explorer to begin editing it.

In the main section of the Visual Studio editor, you are again editing *Program.cs*. Click on the line underneath the text `// write your code here`.

Now, type the following:

```
OutputPort led = new OutputPort(Pins.ONBOARD_LED, false);❶
InputPort button =
  new InputPort(Pins.ONBOARD_SW1, false, Port.ResistorMode.Disabled);❷
bool buttonState = false;❸
```

❶ As with the previous example, the first line of code creates an `OutputPort` so that you can turn the onboard LED on and off.

❷ The second line of code creates an *InputPort*. An InputPort lets you read the voltage level of the pins on the Netduino (or in this case, the voltage coming from the pushbutton). `Pins.ONBOARD_SW1` is shorthand that tells the Netduino which pin of the microcontroller to use for input. The second value, false, tells the runtime that you don't need glitch filtering (if you enable it, Netduino will take multiple readings during a button press to make sure the reading is correct). The final value, `Port.ResistorMode.Disabled` indicates that Netduino won't use a built-in resistor to affect incoming signals on the microcontroller's digital pin.

NOTE: I explore the resistor mode in greater detail later in "Pushing the MakerShield's Button" on page 34.

❸ The third line of code creates a variable named `buttonState`. A variable is a way to store and manipulate data in the memory of the Netduino. In this case, the app stores whether or not the Netduino's pushbutton is currently being pushed in the `buttonState` variable. The word *bool*, which precedes the variable, indicates that `buttonState` will store a Boolean value. Boolean values are values that are true or false, perfect for this application. Finally, `= false` sets the state of the value to false, by default. It's generally a good idea to set variables to a default value so that you make your intentions clear to anyone else (including you!) who reads the source code in the future.

Now you're going to read the state of the pushbutton and turn the LED on and off as the pushbutton is pushed and released. First, create an infinite loop as before. Add the following code to your project:

```
while (true)
{
}
```

Then, read the current state of the pushbutton and write out that state to the LED. Between the two sets of curly braces, insert the following two lines of code:

```
buttonState = button.Read();
led.Write(buttonState);
```

Your final program's `Main()` method should look like this:

```
public static void Main()
{
    // write your code here
    OutputPort led = new OutputPort(Pins.ONBOARD_LED, false);
    InputPort button =
      new InputPort(Pins.ONBOARD_SW1, false, Port.ResistorMode.Disabled);

    bool buttonState = false;

    while (true)
    {
        buttonState = button.Read();
        led.Write(buttonState);
    }
}
```

Now, you're almost ready to run the app, press the pushbutton, and use it to control the state of the LED. But first, you need to make sure you're deploying the project to the Netduino instead of to the built-in emulator. Click on the Project menu and select your project's properties. Then click on the .NET Micro Framework category on the left side. Change the Transport from Emulator to USB, and then make sure that the Device selection box shows your Netduino.

Now run your project. Press the Start Debugging button in the toolbar at the top of the screen or press F5.

After a few seconds, your Netduino app will be running. Once the blue LED turns off, you know that your board has booted. Press the pushbutton, and the blue LED will turn on. Release the pushbutton, and the blue LED will turn back off. Congratulations!

NOTE: The Netduino's pushbutton uses a special wiring configuration to enable it to act as both a reset button and a digital input. It also technically sends a low voltage when pushed even though your code will see a value of `true`. These values are reversed for the pushbutton inside the Netduino firmware; they are logical values instead of physical values.

Early versions of the Netduino firmware did not reverse the physical values. If your Netduino's LED exhibits the reverse behavior when you run this sample, you can either update your firmware or change the code from:

```
buttonState = button.Read()
```

to:

```
buttonState = !button.Read()
```

The exclamation mark before the word `button` reverses the result.

Conclusion

You've now created, deployed, and run your first Netduino projects. You can unplug the Netduino from your computer and demonstrate your first projects to others.

7/Sound and Motion

Speakers generate sounds by vibrating.

You can generate those sound vibrations with the same PWM feature you used in Chapter 6 to adjust the intensity of LEDs. By changing the period and length of pulses, you can change the frequency of the sounds. And by using pulses that correspond to musical notes, you can play songs.

You can also use the length of pulses as a signal to a servo (motor), setting the servo to certain angular positions through specific pulse lengths.

Making Music

To play a song, you'll need a speaker. For this example, use a common and inexpensive piezo speaker.

Wire up the piezo's positive wire to pin D5 on your Netduino. Wire up the piezo's negative (ground) wire to the GND pin on your Netduino. Figure 7-1 shows the connections.

NOTE: Some piezos have legs rather than wires. In most cases, one leg is longer than the other, and the long leg is positive. There may also be faint markings on the piezo indicating which is positive and which is negative. If in doubt, check your piezo's instructions or datasheet for proper wiring.

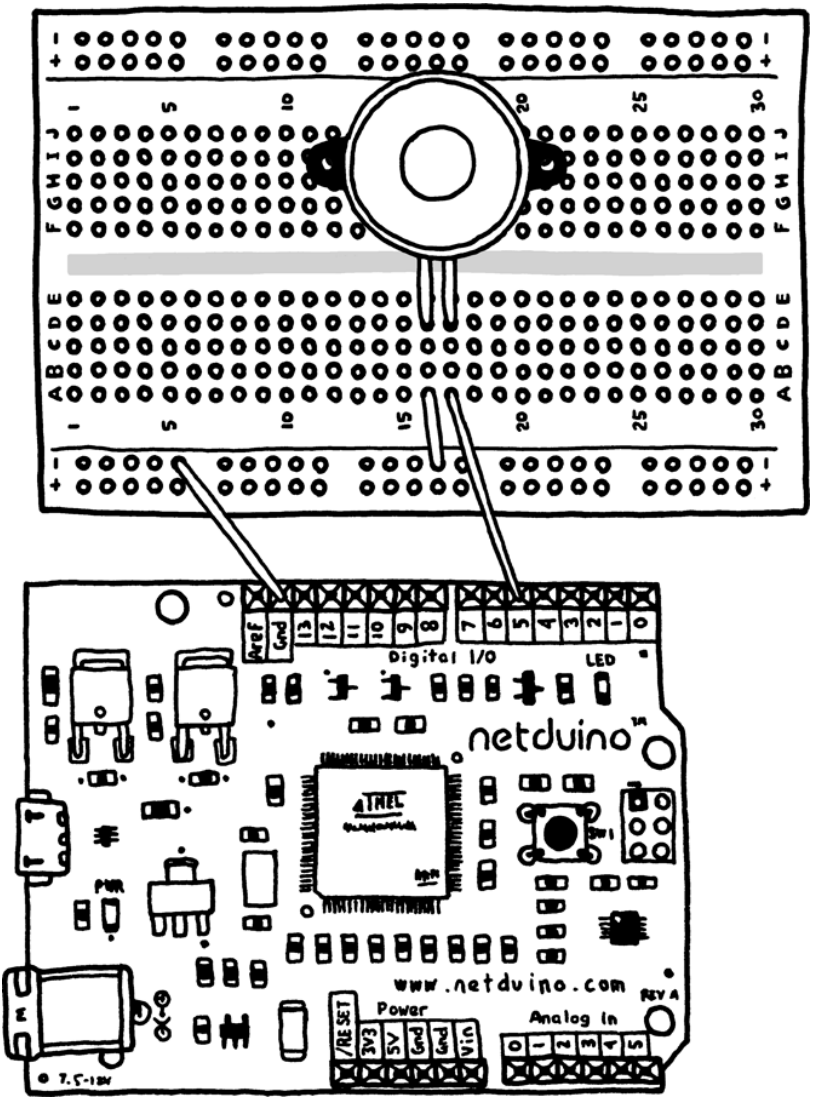


Figure 7-1. *Connecting the piezo*

Create a new Netduino project. Add the following code to the top of your Main routine:

```
// store the notes on the music scale and their associated pulse lengths
System.Collections.Hashtable scale = new System.Collections.Hashtable();❶
```

```
// low octave
scale.Add("c", 1915u);❷
scale.Add("d", 1700u);
scale.Add("e", 1519u);
scale.Add("f", 1432u);
scale.Add("g", 1275u);
scale.Add("a", 1136u);
scale.Add("b", 1014u);

// high octave
scale.Add("C", 956u);
scale.Add("D", 851u);
scale.Add("E", 758u);

// silence ("hold note")
scale.Add("h", 0u);
```

❶ I'm introducing a new concept here called a `Hashtable`. A `Hashtable` is a collection of keys and values. The keys here are the notes, and the numbers are the pulse length values. `Hashtable`s make it easy to look up values by specifying their keys. You'll do this later by looking up notes and getting back their pulse length values.

❷ The "u" after each value signifies that the values should be stored as unsigned integers (whole numbers). You could store this data in other ways, but this makes the sample simpler to understand later on.

NOTE: These note values are calculated by the equation *1 / (2 * toneFrequency)*. toneFrequency is the number of cycles per second for a specific note. You can extend this scale further if desired.

Next, specify the speed of the song. You'll specify this in beats per minute, a common measurement for music.

```
int beatsPerMinute = 90;
int beatTimeInMilliseconds =
  60000 / beatsPerMinute; // 60,000 milliseconds per minute
int pauseTimeInMilliseconds = (int)(beatTimeInMilliseconds * 0.1);
```

To calculate the beatTime (length of a beat) in milliseconds, divide 60,000 (the number of milliseconds in a minute) by the number of beats per minute.

And since you want a tiny bit of pause in between each note (so that the notes don't all blur together), multiply the beat length by 10% to create a pause length. Later on, you'll subtract 10% from the beat length and insert this pause instead, keeping a steady rhythm but making the song more pleasant.

NOTE: You *casted* the pause time to an integer value. Integers are positive or negative whole numbers. Since 0.1 (10%) is a fractional number and not a whole number, you need to do this explicitly. The programming language is smart and realizes that you might lose a little bit of accuracy by ending up with a few tenths of a number that can't be stored in an integer. So it requires that you explicitly *cast* the product into an integer to show you know that you might lose some precision.

Now, define the song. This song involves animals on a farm and the sounds that they make:

```
// define the song (letter of note followed by length of note)
string song = "C1C1C1g1a1a1g2E1E1D1D1C2";
```

The song is defined as a series of notes (upper-case for higher octave, lower-case for lower octave) followed by the number of beats for each note. For the musicians' reference, one beat is a quarter note in this instance and two beats is a half note, in 4/4 time.

Now that you have a musical scale and a song, you need a speaker to play them. You'll add one here:

```
// define the speaker
PWM speaker = new PWM(Pins.GPIO_PIN_D5);
```

You're all set to play the song. Create a loop that reads in each of the note/length entries in the song string, and then plays them by generating specific pulses for the piezo using PWM.

Create a loop:

```
// interpret and play the song
for (int i = 0;❶ i < song.Length;❷ i += 2❸)
{
    // song loop

}
```

❶ The first part of the for statement, `int i = 0`, establishes the variable `i` as a position counter in the encoded song's string.

❷ The middle part of the statement, `i < song.Length`, repeats the loop until you've finished playing the song.

❸ The third part of the statement, `i += 2`, moves the position forward by two places (the size of a note/length pair) each time through the loop.

Inside this loop, you'll extract the musical notes and then play them. Inside the pair of curly braces, add the music interpreter/player code:

```
// extract each note and its length in beats
string note = song.Substring(i, 1);❶
int beatCount = int.Parse(song.Substring(i + 1, 1));❷
```

❶ The first line reads the note out of the song string at the current position, `i`. `Song.Subtring(i, 1)` means "the string data starting at position `i`, length of 1 character."

❷ The second line reads the beat count of the song string. It reads the beat count at one position beyond the note. Then, it uses `int.Parse()` to translate this letter into an integer number that you can use. This is needed because you can't do math with letters, and you'll need to do a bit of math with the beat count shortly.

Now that you have the note, look up its duration:

```
// look up the note duration (in microseconds)
uint noteDuration = (uint)scale[note];
```

By passing in the letter (the variable `note`), you get back the `noteDuration`. Lookup is a powerful feature of C# that can be used for many purposes.

Whenever you retrieve a value from a collection (a `Hashtable` in this instance), you need to explicitly cast it since the programming language wants to be sure it knows what it's getting.

Now, play the note:

```
// play the note for the desired number of beats
speaker.SetPulse(noteDuration * 2, noteDuration);
Thread.Sleep(
  beatTimeInMilliseconds * beatCount - pauseTimeInMilliseconds);
```

`SetPulse` is another PWM feature (in addition to `SetDutyCycle`, which you saw in Chapter 6). It sets a period of twice the `noteDuration`, followed by a duration of `noteDuration`. This will generate a pulse at the frequency required to sound the desired note.

You use `Thread.Sleep` to keep the note playing for the specified number of beats. You subtracted `pauseTimeInMilliseconds` from the note's sounding time so that you can pause a tenth of a beat and make the music more pleasant.

Now that you've played the note, go ahead and insert that one-tenth beat pause:

```
// pause for 1/10th of a beat in between every note.
speaker.SetDutyCycle(0);
Thread.Sleep(pauseTimeInMilliseconds);
```

By setting the duty cycle to zero, you turn the piezo off momentarily. This creates the nice pause.

Finally, skip past the curly brace that ends the for loop and add one more line of code:

```
Thread.Sleep(Timeout.Infinite);
```

This will pause the program after playing the tune.

The final code should look like this:

```
public static void Main()
{
    // write your code here

    // store the notes on the music scale and their associated pulse lengths
    System.Collections.Hashtable scale = new System.Collections.Hashtable();

    // low octave
    scale.Add("c", 1915u);
    scale.Add("d", 1700u);
    scale.Add("e", 1519u);
    scale.Add("f", 1432u);
    scale.Add("g", 1275u);
    scale.Add("a", 1136u);
    scale.Add("b", 1014u);

    // high octave
    scale.Add("C", 956u);
    scale.Add("D", 851u);
    scale.Add("E", 758u);

    // silence ("hold note")
    scale.Add("h", 0u);

    int beatsPerMinute = 90;
    int beatTimeInMilliseconds =
      60000 / beatsPerMinute; // 60,000 milliseconds per minute

    int pauseTimeInMilliseconds = (int)(beatTimeInMilliseconds * 0.1);

    // define the song (letter of note followed by length of note)
    string song = "C1C1C1g1a1a1g2E1E1D1D1C2";

    // define the speaker
    PWM speaker = new PWM(Pins.GPIO_PIN_D5);

    // interpret and play the song
    for (int i = 0; i < song.Length; i += 2)
    {
```

```
            // extract each note and its length in beats
            string note = song.Substring(i, 1);
            int beatCount = int.Parse(song.Substring(i + 1, 1));

            // look up the note duration (in microseconds)
            uint noteDuration = (uint)scale[note];

            // play the note for the desired number of beats
            speaker.SetPulse(noteDuration * 2, noteDuration);
            Thread.Sleep(
              beatTimeInMilliseconds * beatCount - pauseTimeInMilliseconds);

            // pause for 1/10th of a beat in between every note.
            speaker.SetDutyCycle(0);
            Thread.Sleep(pauseTimeInMilliseconds);
        }

        Thread.Sleep(Timeout.Infinite);
    }
```

Now, run your app. Sing along with the song if you'd like. Look up the full song online and add the rest if desired.

Motors and Servos

There are several types of electromechanical devices that generate motion. Two of the most common are motors and servos.

Motors work by using the PWM DutyCycle, just like LEDs. You can use a motor shield or design your own circuitry to connect motors to your Netduino. Just like LEDs, when you increase the DutyCycle motors spin faster; when you decrease the DutyCycle, they slow down.

Circuitry to power motors generally uses an H-Bridge, which is effectively a device that uses the PWM output of your Netduino to supply a much larger amount of power at a voltage proportional to your PWM signal's "on time." The PWM signal is combined with a GPIO input specifying whether the motor should spin forward and backward. All together these create a negative or positive analog voltage that spins the motor.

The drawback of using motors is the lack of precision. You can spin them faster or slower, but often you want something to move at a specific speed or move to a specific position. When you want to move between positions, you want to use servos.

Servo Control

Servos create motion by turning forward or backward to specific positions. To demonstrate, I'll show you a program that moves through various points of a servo's position range.

To wire up the servo, connect its red wire to the 5V header on your Netduino and its GND wire (usually black, but it might be brown) to a Ground header. Then, connect the signal wire (usually white, but may be another color) to pin D5. Figure 7-2 shows the connections. You'll use PWM signals to specify the servo's position.

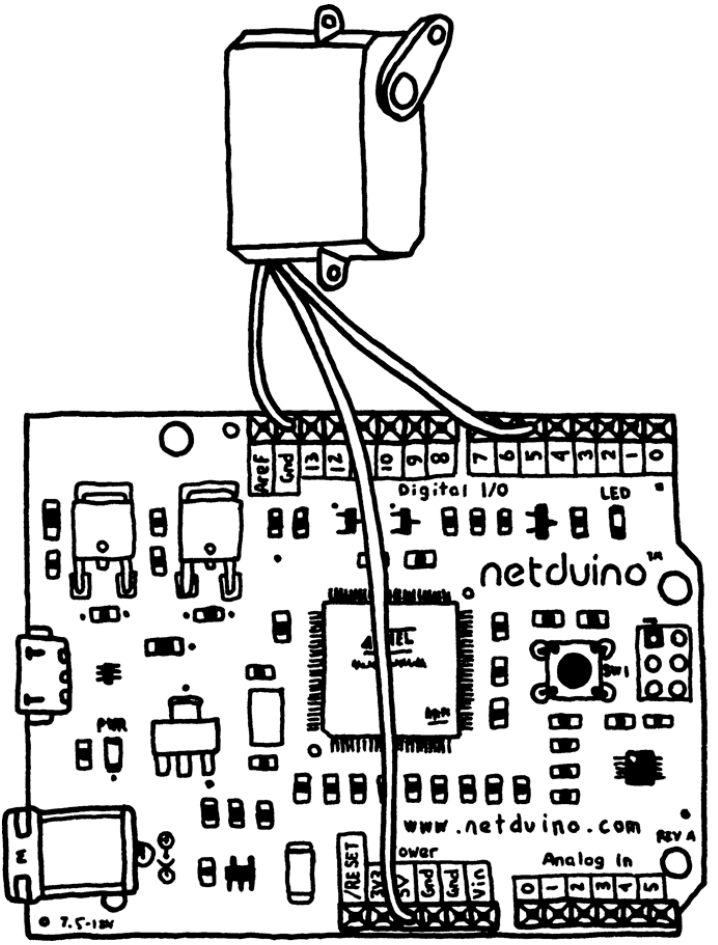


Figure 7-2. *Connecting the servo*

Create a new Netduino project. Add the following code:

```
PWM servo = new PWM(Pins.GPIO_PIN_D5);
```

As before, this creates a new PWM object using pin D5. You'll use specific pulse lengths to direct the servo to move to specific positions.

Now, set some boundaries for the servo. The following values are the number of microseconds in a pulse, which represent the lowest and highest position values for the servo. This will represent a full 90 degrees of motion:

```
uint firstPosition = 1000;
uint lastPosition = 2000;
```

NOTE: Some servos offer extended range, such as 180 degrees. These servos may use the range of 0-1,000 to represent the extra 90 degrees of range.

Next, move through the full range of motion for the servo. Add the following code:

```
// move through the full range of positions
for (uint currentPosition = firstPosition;
     currentPosition <= lastPosition;
     currentPosition += 10)❶
{
        // move the servo to the new position.
        servo.SetPulse(20000, currentPosition);❷
        Thread.Sleep(10);
}
```

❶ The for loop is similar to the one you used for the piezo earlier in the chapter. But instead of using `i` as the counter variable, you're using `currentPosition`. It is good programming practice to give variables descriptive names like this (although `i` is commonly used for a counter variable, so it's okay that it wasn't as descriptive as this one).

You loop from the first position to the final position, at 10 unit intervals (approximately 0.9 degrees each).

❷ Inside the loop, you move the servo to the new position and then sleep for ten milliseconds. This will give you a smooth motion.

Now, return the servo to its first position. You can move the servo backward just as easily as forward:

```
// return to first position and wait a half second.
servo.SetPulse(20000, firstPosition);
```

Finally, finish up the app by putting it to sleep:

```
Thread.Sleep(Timeout.Infinite);
```

Your final app should look like this:

```
public static void Main()
{
    // write your code here
    PWM servo = new PWM(Pins.GPIO_PIN_D5);

    uint firstPosition = 1000;
    uint lastPosition = 2000;

    // move through the full range of positions
    for (uint currentPosition = firstPosition;
         currentPosition <= lastPosition;
         currentPosition += 10)
    {
        // move the servo to the new position.
        servo.SetPulse(20000, currentPosition);
        Thread.Sleep(10);
    }

    // return to first position and wait a half second.
    servo.SetPulse(20000, firstPosition);

    Thread.Sleep(Timeout.Infinite);
}
```

Now run your app. You can now create motion digitally!

This specific demo may remind you of the action of an old typewriter or dot matrix printer; for those not acquainted with either, they are featured in many older movies.

NOTE: Did you notice that the power LED on the Netduino flickered during servo movement? If so, that's because you connected a servo directly to Netduino's 5V supply power header. Servos use a lot of power—especially when they meet resistance—so you'll usually want to give them a separate power supply. If you do this, you may need to connect the ground (GND) pins from both power supplies together.

Conclusion

You have now made music with a piezo and made motion with a servo. It's amazing how many electronics can be controlled through simple pulses of electricity.

Next up: connecting your Netduino Plus to the Internet.

About the Author

Chris Walker is the inventor of the Netduino, host of the Netduino user community, and an expert on .NET Micro Framework.