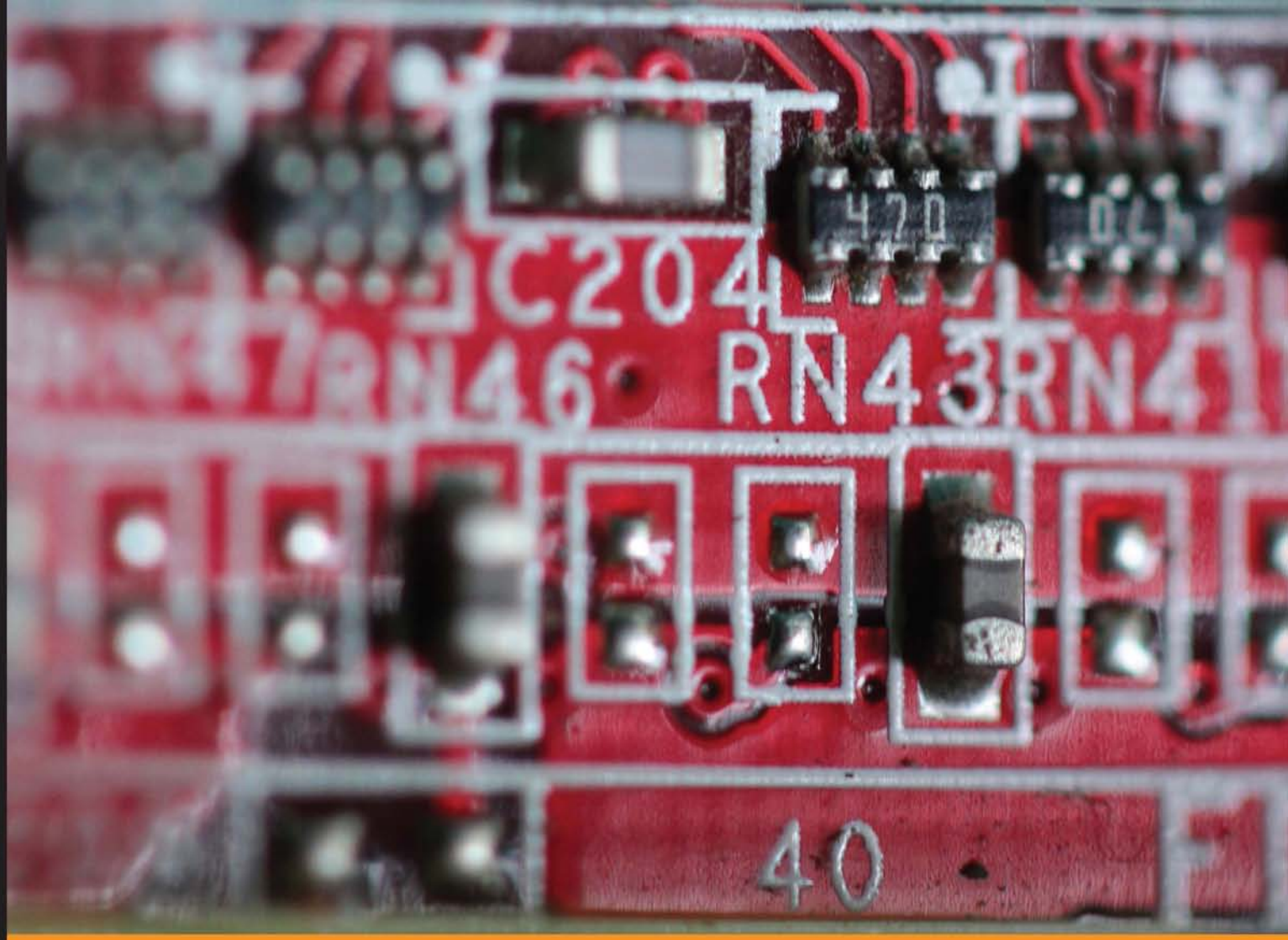


Community Experience Distilled

Netduino Home Automation Projects

Automate your house, save lives, and survive the apocalypse with .NET on a Netduino!

Matt Cavanagh

[PACKT] PUBLISHING

Netduino Home Automation Projects

Automate your house, save lives, and survive the apocalypse with .NET on a Netduino!

Matt Cavanagh

[PACKT] PUBLISHING

BIRMINGHAM - MUMBAI

Netduino Home Automation Projects

First published: August 2013

Production Reference: 1160813

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-84969-782-8

www.packtpub.com

Cover Image by Aniket Sawant (aniket_sawant_photography@hotmail.com)

Credits

Author
Matt Cavanagh

Reviewers
Oliver Manickum
Lance McCarthy

Acquisition Editor
Edward Gordon

Commissioning Editor
Amit Ghodake

Technical Editors
Sharvari H. Baet
Menza Mathew

Project Coordinator
Akash Poojary

Proofreader
Bernadette Watkins

Indexer
Monica Ajmera Mehta

Production Coordinator
Shantanu Zagade

Cover Work
Shantanu Zagade

About the Author

Matt Cavanagh is a professional maker and tinkerer. From his armchair in Durban, South Africa, he lives the dream, wearing suit and slippers with cigar in his hand (fine, just the slippers). He gave up corporate development to start his own company writing Windows and Windows Phone apps, but mostly messes around all day with Netduinos, 3D printers, and his pet quadrocopter. He is also a Nokia Developer Champion who occasionally gives talks on Windows Phone and Netduino.

Oh, and he writes stuff too.

I would like to thank Secret Labs for being awesome, and the Netduino community for being an endless source of ideas and help. Most of all though, I need to thank my beautiful wife, Kaila, for putting up with most of our house being covered in wires and servos.

About the Reviewers

Oliver Manickum has been developing enterprise level software for the past 17 years. He has developed applications across multiple platforms extending from Linux to Windows. He is very passionate about embedded software development, especially open source electronic platforms, where he spends many hours playing with the Arduino and Netduino prototype boards. In real life, Oliver's occupation is developing mobile applications for Android and Windows Phone.

Lance McCarthy is a professional XAML developer, XAML Support Specialist at Telerik, and a Nokia Developer Ambassador for the Northeastern United States. He is an award winning Windows Phone developer and has been writing for the platform since September 2010. Passionate about melding hardware and software, in his spare time Lance uses the power of the Netduino and Windows Phone to create projects that blend the worlds of electronics and human interaction.

I would like to thank Matt Cavanagh for his pure awesomeness, my employers for encouraging creativity, and my wife Amy for her undying patience with my mad scientist moments.

---

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit `www.PacktPub.com` for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

`http://PacktLib.PacktPub.com`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read, and search across Packt's entire library of books.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via web browser

Free access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

This is dedicated to my father, Brett, who is solely responsible for my technology addiction.

I would like to thank Secret Labs for being awesome, and the Netduino community for being an endless source of ideas and help. Most of all though, I need to thank my beautiful wife, Kaila, for putting up with most of our house being covered in wires and servos.

Table of Contents

Preface

When I was a kid I had a Meccano set, which is pretty much the coolest childhood "toy" that exists because it allows you to build things without actually knowing how to build things. But it was limited—there was no interactivity beyond a switch and a motor.

So I started pulling apart appliances such as Hi-Fis and putting them back together. I got so good at doing this that, at one stage, was able to put things back together with only a small handful of screws and parts left over.

But there was a very definite gap between what I wanted to do and what I could actually do with my limited knowledge. Child-oriented electronics kits were pretty useless too, as they would help you make one or two very specific projects, but didn't equip you for anything beyond that.

Jump forward to today, and finally there is something to bridge that gap. For me, that is the Netduino. The Netduino (and other development boards like it) allows everyone from a hobbyist to an electronic engineer to build pretty much anything, and removes all the complexities that were holding us back before. You get all of the power, and a lot less of the frustration.

This book will guide even a novice .NET developer through a range of projects specifically chosen to cover all the fundamentals of the platform. With the knowledge gained from these projects, you will be able to build a massive range of gadgets—only limited by what you can think up, not just by the index page of this book. It will even give you a jumpstart into developing apps for the AGENT smartwatch, which is a watch running the .NET Micro Framework, developed by the team that made the Netduino.

When you've completed the book, head on over to `http://blog.roguecode.co.za`, where I regularly blog about more advanced topics usually related to Netduino and Windows Phone.

With the rise of electronic development boards, 3D printing, homemade laser cutters, and much more, being a maker is cool now. So go forth and conquer.

What this book covers

Chapter 1, *Getting Started with Your New Toy*, explains installing the software to use your Netduino, getting it connected, and making sure it is updated.

Chapter 2, *Lights, Camera, Action – Sound-controlled Ambient LEDs*, explains controlling ambient light intensity with sound, making movie time that bit more awesome.

Chapter 3, *Get Connected – Bluetooth Basics*, explains how to connect your mobile phone to your Netduino to control your projects.

Chapter 4, *Let There Be Light, by Clapping or Tapping*, explains turning lights (and other appliances) on and off by using sound.

Chapter 5, *Honey, I'm Home – Automated Garage Doors with Your Mobile Phone*, explains how to open and close your garage doors using your mobile phone over Bluetooth.

Chapter 6, *You've Got Mail, and Here's a Flag to Prove It*, explains how to get your Netduino to raise a flag when you have a new e-mail—and display a preview on a screen.

Chapter 7, *I'm Completely Dude, Sober – a Homemade Breathalyzer*, explains how to make a breathalyzer at home.

Chapter 8, *Hide Yo' Kids, Hide Yo' Wife – with Automated Locks,* explains how to use a keypad and pin to lock/unlock your doors.

Chapter 9, *Rogue Alert – Detect Intruders in Your House or Fridge*, explains how to make your Netduino e-mail you when motion is detected in your house, fridge, or cupboards, and when doors are opened, or a tripwire is tripped.

Chapter 10, *Saving Lives – Automated Watering*, covers automatically watering your plants or filling up your pets' water bowls when the soil becomes dry or their water gets low.

What you need for this book

The following hardware and software is required to follow the examples given in the book:

- Visual C# Express 2010 or Visual Studio 2010 or above
- .NET Micro Framework SDK Version 4.2
- Netduino SDK Version 4.2.2.0

Who this book is for

This book is ideally suited for a lazy person who has some experience in C#, and has used a Netduino before but wants to explore more advanced topics. However, the book starts from the very basics so can be picked up even by novices.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "Now add `using Toolbox.NETMF.NET;` to the top of Program.cs."

A block of code is set as follows:

```
private static void btn_OnInterrupt(uint data1, uint data2,
  DateTime time) {
    HandlePress('*');
}
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes, for example, appear in the text like this: "Open up Visual Studio and create a new **Netduino Plus 2 Application**."

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
Getting Started with Your New Toy

Congratulations on becoming the proud new owner of a shiny little Netduino Plus 2! But before devising a plan to take over the world, you will probably want to get it plugged in and updated. Keeping both the Netduino firmware and the software on your PC updated is paramount to world domination.

In this chapter we will cover:

- How and what software to install to be able to write programs for the Netduino
- Getting your Netduino's firmware up-to-date
- Writing your first Netduino program

To find each of the files required, navigate to `http://bit.ly/LazyDownloads`. You should install everything in the order laid out in the following sections or the world may potentially end.

This book uses the latest versions available at the time of writing. This means that, by the time you read this, there could be a newer update. You can keep the software up-to-date with the latest betas by heading over to the Netduino forums (`http://forums.netduino.com/`) and visiting the **General Discussion** section. Alternatively, if you prefer to use only final (nonbeta) software, then you should get your software from `http://bit.ly/LazyNetduinoDownloads`.

Prerequisites

While it is possible to develop for the Netduino on OSX and Linux, in this book we only cover Windows. In terms of hardware, most of the projects can be done on any Netduino model, but we will be focusing on the Netduino Plus 2 because it is the latest and greatest version.

Visual Studio

To write all the code, we will be using C# with Visual Studio as the IDE. If you've never used Visual Studio before, no problem, it's very simple and powerful. But best of all, the Express edition is free (free as in beer)! You can use either Visual C# 2010, or Visual Studio 2012 Express for Windows Desktop. We will be using the latter, which requires Windows 7 or above.

Download the ~600 MB `.iso` file (or the install package). If you have downloaded the `.iso` file, then right-click on the file and click on **Mount**. A window will pop up showing the contents of the disc image. Run the `wdexpress_full.exe` file.

After the installation is done, you can launch it. You will be prompted to enter a serial key, which you can get for free by registering, or you can use it without a serial key for 30 days.

The .NETMF SDK and the Netduino SDK

To develop using Netduino, you will need both the **.NET Micro Framework SDK** and the **Netduino SDK** installed. The former is an open source framework which is a subset of .NET developed by Microsoft to make it easier to write code for low-powered embedded devices. The Netduino SDK is built on top of that to add Netduino-specific functionality and helper functions. Follow these steps to install the required software:

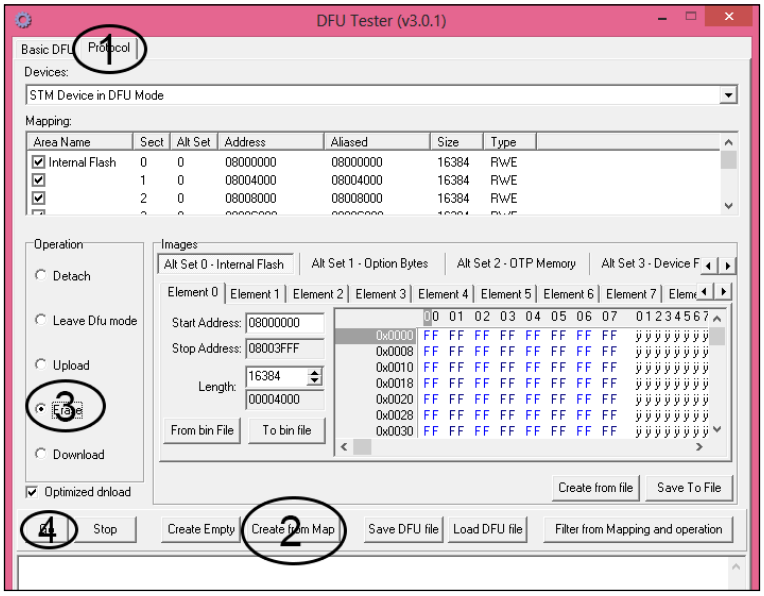
1. Download the ~20 MB .NET Micro Framework SDK and install it. Select the **Complete** option when prompted.
2. Download the ~10 MB Netduino SDK and install it. Accept the permission prompts.
3. Once all are successfully installed, it's usually a good idea to restart Windows.
4. You can now plug in the Netduino with the supplied USB cable! The first time you do this, Windows will install drivers for it automatically.

The Netduino firmware

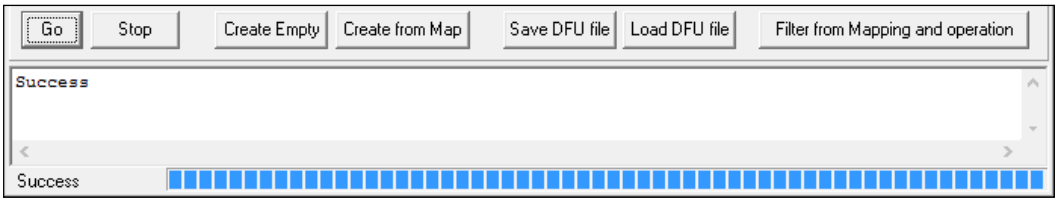
The final step before being able to use your Netduino is to update its firmware. Currently, this latest firmware is beta (v4.3.0.0 beta 1), but should be final soon. Although you don't have to update it, it is recommended that you do. Whether you use the beta firmware, or the final version, the following steps will show you how to get it onto the Netduino:

The following steps are an expansion of the ones over on the Netduino Forums: `http://bit.ly/LazyBetaFirmware`

1. You need to get the Netduino into the bootloader mode. So with it unplugged, press and hold the push button (this is located just below the **netduino plus 2** text seen on the board), then plug in the USB cable to your computer. There should now be two lights glowing on the Netduino, one white and one blue. You can release the button once it is plugged in.
2. Download and install STDFU Tester v3.0.1 from `http://bit.ly/LazyDFUSE` and launch the application from your start menu.
3. Follow these steps to first erase the current firmware:
    1. Click on the **Protocol** tab near the top.
    2. Click on the **Create from Map** button near the bottom.
    3. Check the **Erase** radio button on the left.
    4. Finally, click on the **Go** button at the bottom left.



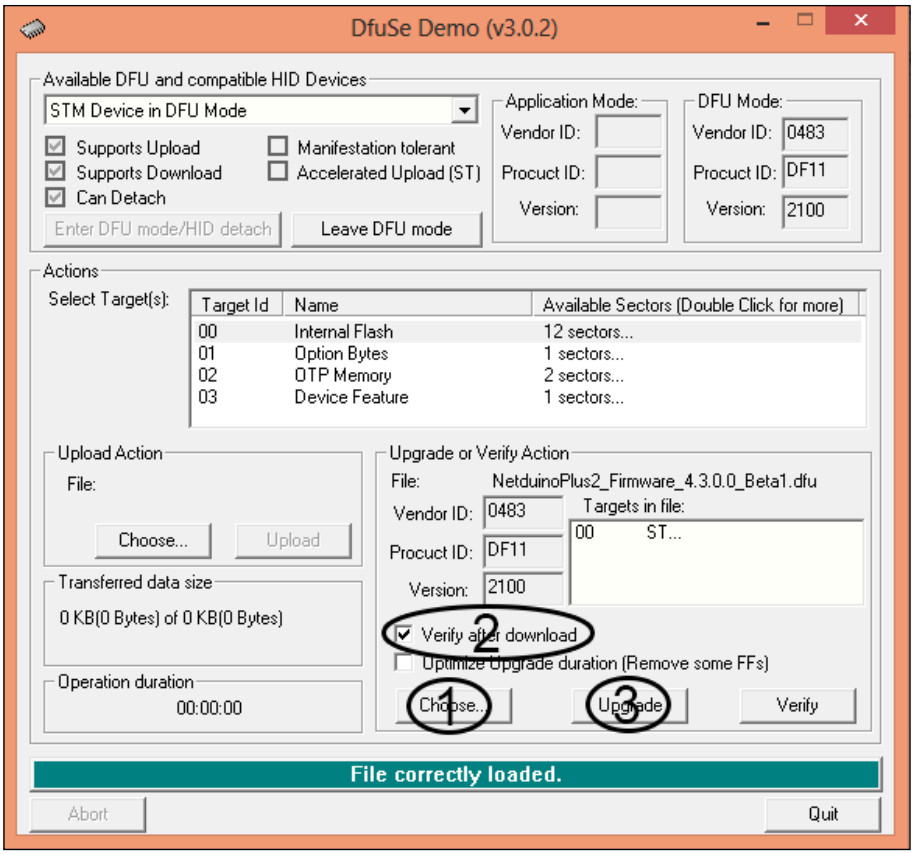
4. A progress bar will appear at the bottom of the application, and after a few seconds **Success** will be displayed in the bottom block. Close the application.



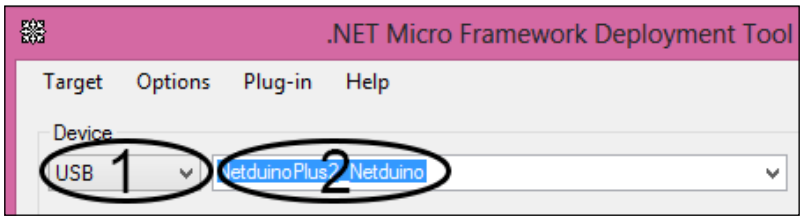
5. Along with STDFU Tester, an application called DfuSe Demonstration would have been installed; launch it.
6. The firmware file you download will be a zipped archive, so unzip it to a new folder. Inside should be a single file named something like `NetduinoPlus2_Firmware_4.3.0.0_Beta1.dfu`.
7. To get the firmware onto your Netduino follow these steps:
    1. Click on **Choose** in the bottom right pane and locate the `.dfu` file from step 6.
    2. Select the **Verify after download** checkbox above the **Choose** button.
    3. Click on **Upgrade** available to the right of the **Choose** button.
    4. Click on **Yes** in the message that appears saying **Your device was plugged in DFU mode**.
    5. A progress bar at the bottom will show you that it is busy working, and when complete should display a message **Verify successful**.
    6. Close the application.

Downloading the example code

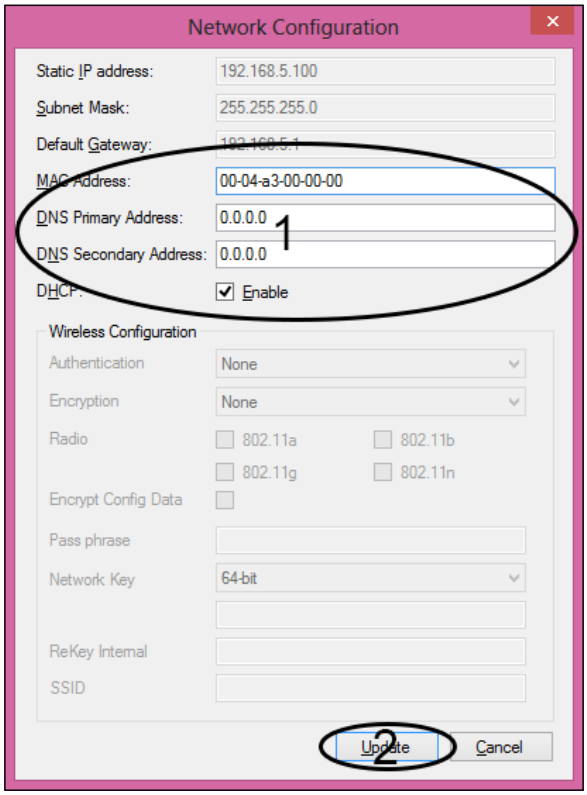
You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the fi les e-mailed directly to you.



8. Unplug the Netduino and then plug it in again (don't hold the button this time) to get it out of the bootloader mode. The blue light on the board should turn off after a few seconds.
9. Now open `MFDeploy.exe` from `C:\Program Files (x86)\Microsoft .NET Micro Framework\v4.3\Tools\MFDeploy.exe` (remove the (`x86`) bit if you are using a 32-bit version of Windows).
10. In the drop-down menu near the top change **Serial** to **USB**. Then make sure your Netduino is selected just to the right.



11. From the top menu, navigate to **Target | Configuration | Network**. Enter your DNS settings (usually the IP address of your home router) and **MAC Address** (which can be found on the sticker underneath your Netduino). Select DHCP if your router has that enabled, or set an IP manually. Then click on **Update** as shown in the following screenshot:



12. Close the application.

Hello world

Before getting onto the more exciting projects, let's do a very basic one just to make sure you know where to put the code, and to check whether your Netduino is set up correctly. We are going to make the onboard LED (the blue one) fade on and off. Technically, you can't actually make an LED fade, but you can make it look like it is turning it on and off very quickly. If the LED is on for more time than it is off then the light will appear bright, but will appear dim if the LED is off for more time than it is on. The onboard LED of the Netduino 1 does not support PWM (which is used to fade), so will be impossible to use here.

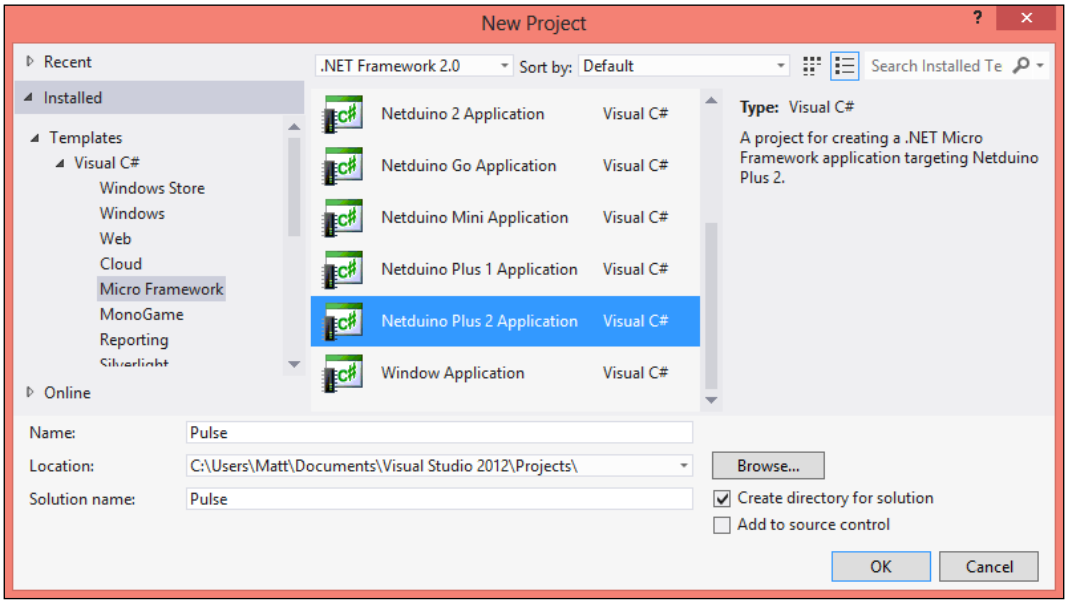
Things you need

All you need for this project is your Netduino.

The first project

Follow these steps to create our first project:

1. Open up Visual Studio 2012 (or whatever version you are using) and navigate to **File** | **New** | **Project**.
2. Find the **Netduino Plus 2 Application** template by navigating (in the left pane) to **Installed** | **Templates** | **VisualC#** | **MicroFramework**. This template is the one which we will use for every project for the Netduino. At the bottom of the window, fill the **Name** and **Location** textbox with a suitable name and location for the project, then click on **OK** to create the project, as shown in the following screenshot:



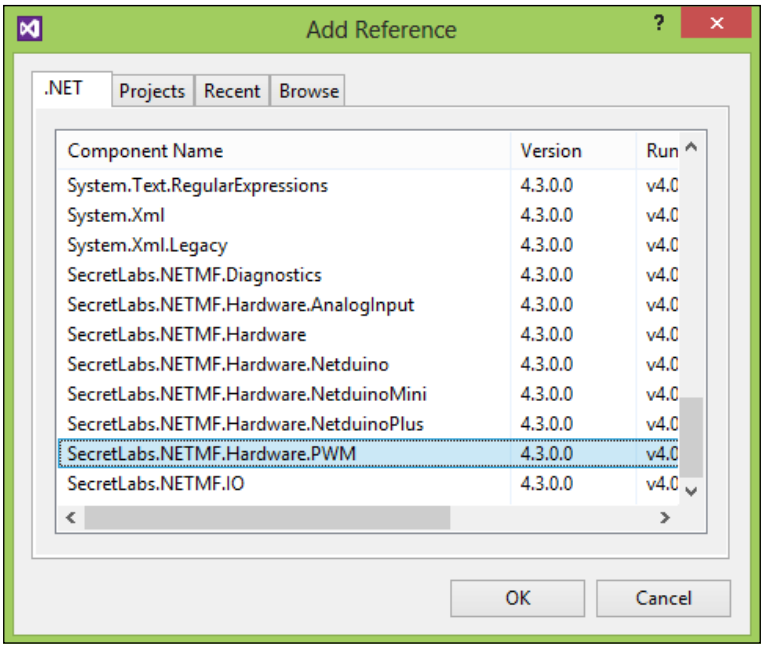
3. In the **Solution Explorer**, double-click on **Program.cs**. This is the code file that will contain most of the code for our applications. The `Main` method is the one that will run when the application is launched on the Netduino, which is why by default there is a comment there which says **// write your code here**. Generally, you will put a `while` loop within the `Main` method which will loop forever (as long as the Netduino is on) and carry out your bidding.

4. Instead of manually making the LED turn on and off really quickly, we are going to use something called **Pulse-Width Modulation** (**PWM**), which does all the hard work for us. The easiest way to do this is to use the `PWM` class that the Netduino SDK comes with—but the DLL containing it is not included by default. So in the **Solution Explorer** window, right-click on the **Reference** folder and click on **Add Reference**. In the window that appears, make sure you are in the **.NET** tab, then scroll down until you find **SecretLabs.NETMF. Hardware.PWM** and select it. Click on **OK** to add the DLL.

   Now that the DLL has been added, we still need to add a using statement to the top of the code in `Program.cs`. Add this line just below the other using statements:

   ```
   using NPWM = SecretLabs.NETMF.Hardware.PWM;
   ```

   You may notice that this looks slightly different from the lines above it. The reason for adding `NPWM =` is to give this namespace a "nickname" so that we are able to specifically reference this PWM class as opposed to another class with the same name in one of the Microsoft assemblies.



5. Within the `Main` method, we will create a new instance of PWM, and in the constructor we will specify which pin on the board we are referring to (using the Netduino `pin enum`):

   ```
   NPWM led = new NPWM(Pins.ONBOARD_LED);
   ```

Notice that this is using the "nickname" we created earlier.

To set the brightness of the LED, we need to tell the PWM instance how often it should be on compared to off—this is called its **DutyCycle**. DutyCycle is similar to percentage, where 100 is always on, 50 is on for half of the time and off for the other half, and zero is off all the time.

To demonstrate this, add the following code after the instance of PWM that you created:

```
while (true) {
  led.SetDutyCycle(30);
  Thread.Sleep(1000);
  led.SetDutyCycle(60);
  Thread.Sleep(1000);
  led.SetDutyCycle(90);
  Thread.Sleep(1000);
}
```

In the preceding code, we have a `while` loop which will run for infinite time. This will ensure that our code will continue to run forever. We set the `DutyCycle` to `30` (meaning 30 percent brightness) and then tell the code to pause for `1000` minutes (which is a single second) before continuing onto the next line. Basically, this code will set the LED to 30 percent, wait for a second, set it to 60 percent, wait for a second, set it to 90 percent, wait for a second, and finally loop back to the top and start again.

6. Before running the code, we need to tell Visual Studio where to deploy it. Right-click on the project in **Solution Explorer** and select **Properties**. Make sure you are clicking on the project (second item) and not the solution (top item). Next, select **.NET Micro Framework** on the left. In the center pane, you will now need to ensure that USB is selected in the first drop-down list, and your Netduino is selected in the second one. If your Netduino isn't showing up, make sure it is plugged in correctly and that you followed the installation steps correctly.
7. To deploy the code onto your Netduino and run it, click on the **Start** button located at the top-center of Visual Studio (it's the one with the green "play" icon) or press *F5* on your keyboard. It will take a few seconds for it to deploy and run, and then your LED should be changing brightness.

8. Proud of yourself? Good. Now stop the debugging by pressing the red "stop" button available in Visual Studio, or with *Shift* + *F5* on your keyboard. The code will continue running on your Netduino, but Visual Studio will no longer be connected to it. You need to stop debugging in order to change code and redeploy.
9. But that's pretty boring. So to make the light fade in and fade out, delete that code, and insert this:

```
while (true) {
  for (uint p = 0; p < 100; p++) {
    led.SetDutyCycle(p);
    Thread.Sleep(10);
  }
  for (uint p = 100; p > 0; p--) {
    led.SetDutyCycle(p);
    Thread.Sleep(10);
  }
 }
```

This is a little bit more complicated, but nothing you can't handle!

The code has two for loops. In the first one, each iteration will increase the value of `p` by `1`, set the DutyCycle to that value, and then wait for 10 milliseconds. Basically, it will fade the LED in. The second loop does the exact opposite by starting `p` at `100` and decreasing it by `1` each iteration. The end result is that the LED will fade on and off over and over again.

Occasionally your Netduino could become unresponsive. The first thing to try is to just unplug it and then plug it back in and try to run the code again. If that doesn't work then chances are that the code you have previously deployed has crashed in such a way that you cannot interact with the board. To fix this you can use `MFDeploy.exe` to erase the deployment (clear the code).

Summary

In this chapter, you installed all the software that you need to get your Netduino up and running. In doing so, you also learned the process of updating the firmware, which you will need to do each time there is a new release and if you want to keep your Netduino up to date.

Finally, you wrote your first Netduino application and had a wave of pride rush over you as the little onboard LED twinkled to life. With the setup all done, we can get onto the real fun stuff! Code ahoy!

In the next chapter, we will create our first real application—one that has a point.

2

Lights, Camera, Action – Sound-controlled Ambient LEDs

Now that everything is set up and ready to go, we can start the first real project! You, no doubt, love watching movies on your big-screen TVs, with lights off, sound up, and popcorn in hand. But how can we make this more immersive? With LEDs, of course!

We are going to make a little project that uses a microphone to listen to the level (loudness) of sound in the room and effect the brightness of a bunch of LEDs accordingly. When we're done, you can stick this behind your TV and get an awesome glow around the screen during the most intense action scenes.

In this chapter we will cover:

- How to connect up a basic microphone
- Powering components from a separate battery pack while still controlling them from the **Netduino**
- Using transistors in a circuit

Things you need

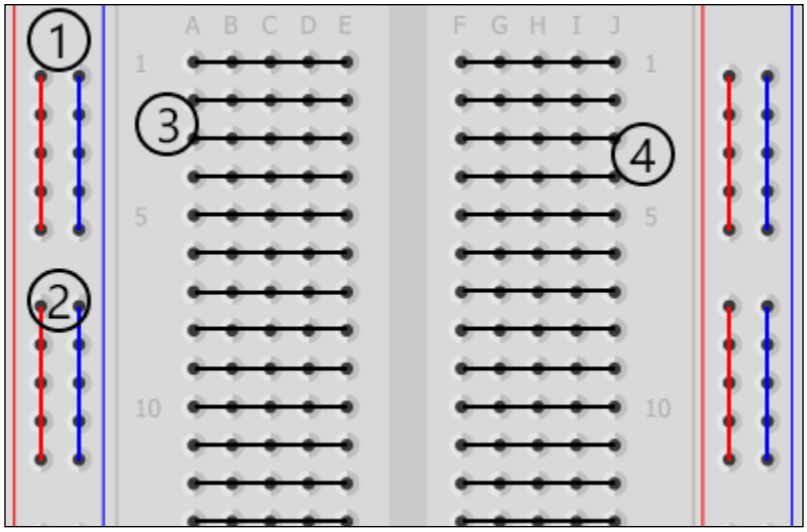
- Netduino
- Breadboard
- 4xAA power supply
- TIP122 transistor (or equivalent NPN transistor)

- Electret microphone (with breakout board)
- 1K ohm resistor
- 68 ohm resistor (one per LED)
- LEDs

The choice of number of LEDs (and color) is totally up to you. In the example, we just use two White LEDs (with a forward voltage of 3.5V, 20mA), but obviously, that will not provide much light. You should aim to have between 10 and 20 LEDs. You may also use whatever type of battery pack you have lying around (or get a 4xAA battery holder). The ohms for the LED resistors will vary based on your LEDs and power supply.

Breadboards

Breadboards are great to prototype your designs before soldering anything, but you need to know how they are laid out. In the diagram below, the solid lines between pins indicate what pins are connected together. The wires you will stick into the breadboard are called Jumper wires, and have solid tips to make them easy to insert. You should buy lots of these:



- **1**: These are power lanes, and will usually have red and blue lines printed indicating positive and negative, respectively. Each pin in a single lane will connect to neighboring pins in the same lane (vertically in the diagram).

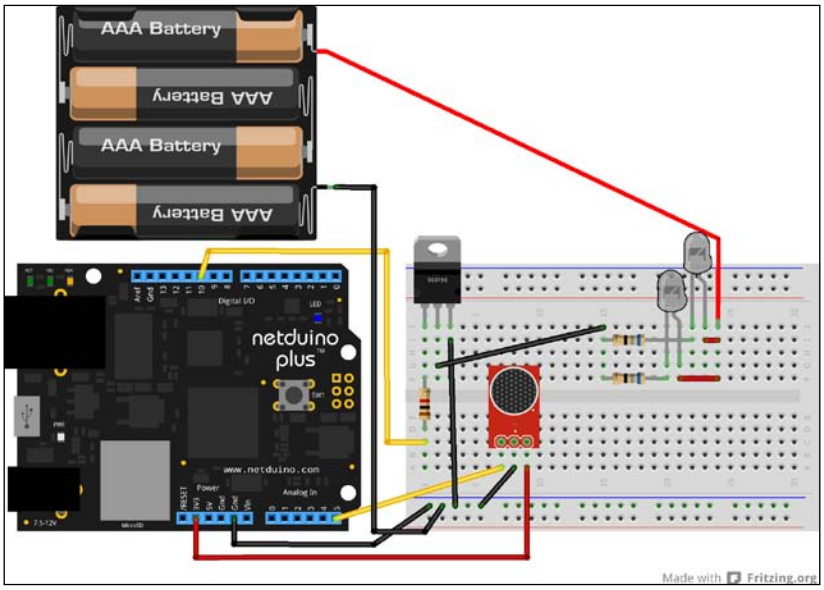
- **2**: If there is a gap in the printed colored line it means they are not connected. You can put a wire over the gap to connect them. If the line has no gaps then they are connected (as is the case in the preceding image).
- **3**: Inner pins work on the same principle, but are perpendicular to the power lanes.
- **4**: Separate sections (indicated by the gap) are not connected.

For further information on breadboards, take a few minutes to read up here: `http://en.wikipedia.org/wiki/Breadboard`.

The project setup

In our circuit we are using a separate battery pack to power the LEDs. It is perfectly possible to turn an LED `ON` and `OFF` by directly connecting it to the Netduino power pins, but that isn't a good idea because once you have multiple LEDs, they may draw more power than a single Netduino pin can supply and could kill it. In between the LEDs and the power supply there is a transistor. In our case, this is used like a fancy dimmer switch controlled by a Netduino PWM pin (pretty much exactly like how we controlled the on-board LED in the first chapter).

Follow the diagram below very carefully to get your circuit correct. You should notice that the ground (negative) wire from the battery pack connects in the breadboard lane with the ground from the Netduino; this is required to be able to control devices from the Netduino while still powering them from a separate power supply.



The microphone has 3 pins. The first one is the signal wire and can be plugged into any of the Netduino **Analog IN** pins (we use pin **5**) so that we can read the sound values. The other two are ground (might be labeled **GND**) and 3.3V (might be labeled **VCC**).

The LEDs are powered by the positive wire from the battery pack, and the negative wire from the center pin of the transistor. In the example diagram there are only 2 LEDs, but you can use as many as you want (within the limits of your power source). Because LEDs vary, you may need to wire yours differently to the diagram. The easiest way to work out how to do this, and what resistors you need, is to head over to `http://ledcalculator.net/` which will draw a wiring diagram for you based on the values you give it. LEDs have two legs, the long one is positive/anode and the short one is negative/cathode.

The Netduino code

1. Just like in the first chapter, you will need to open up Visual Studio, and create a new **NetduinoPlus 2 Application** and name it anything you want.
2. In the **Solution Explorer** right-click on **project** | **Add** | **Class…** then choose the Class file in the **Micro Framework** | **Code** section, call it `LowPassFilter.cs`, and hit **Add**. Change the class to represent the following code:

```
public class LowPassFilter {
  private double _smoothingFactor;
  public double SmoothedValue;
  publicLowPassFilter(double smoothingFactor) {
    _smoothingFactor = smoothingFactor;
  }
  public void Step(double sensorValue) {
    SmoothedValue = _smoothingFactor * sensorValue + (
    1 - _smoothingFactor) * SmoothedValue;
    }
}
```

3. This is a very basic low-pass filter. Why do we need it? When using sensors (microphones, accelerometers, and so on) there is always going to be quite a lot of noise (which refers to "spikes" in the data), so we use a filter to smooth these values out. If we didn't use this, our LEDs would flicker a lot because of the amount of "bad" data that the microphone returns. By smoothing it out and taking more of an average over time, we get a more realistic set of values to work with. To use this class we just create a new instance with a `smoothingFactor` function (a lower number will smooth the data more but have more lag, whereas a higher number will smooth less but be more responsive), and each time we get a new value from a sensor we update it by calling the `Step` method.
4. As before, add a reference to **SecretLabs.NETMF.Hardware.PWM**, along with **SecretLabs.NETMF.Hardware.AnalogInput**.
5. Using the **Solution Explorer** open up **Program.cs**. At the top, add the two "usings" for the classes you just referenced:

```
using NPWM = SecretLabs.NETMF.Hardware.PWM;
using NAnalog = SecretLabs.NETMF.Hardware.AnalogInput;
```

6. Within the **Main** method, insert this code:

```
NAnalogmic = new NAnalog(Pins.GPIO_PIN_A5);
while (true) {
  double reading = mic.Read();
  Debug.Print(reading.ToString());
  Thread.Sleep(5);
}
```

   First we create a new instance of the **AnalogInput** pin which the signal wire of our microphone is connected to. Then, within a loop, we read the value from the pin and print it out to the **Output** window in Visual Studio (**View | Output**). Click on **Start** and the code will be deployed to your Netduino. After a few seconds (when the blue onboard LED turns `OFF`), you should see all the values appearing in the **Output** window. Try blowing on the microphone to see the values change. Take note of the average value when there is silence. Using the `SparkFun` microphone, I get about 535. If you are unsure you can copy a range of the values from the **Output** windows and paste them into **Excel**, then get an average. Delete the code you just added and replace it with this:

```
NPWM led = new NPWM(Pins.GPIO_PIN_D10);
NAnalogmic = new NAnalog(Pins.GPIO_PIN_A5);
LowPassFilter filter = new LowPassFilter(0.03);
doublemaxSound = 535;
```

7. This code creates a new instance of the digital pin that the transistor pin connects to. Also, it creates the low-pass filter with a smoothing value of `0.03`, and declares a variable of the max sound value that we took note of in the previous step. Below that code, we will put the loop that does everything:

```
while (true) {
  doublemicValue = mic.Read();
  micValue = micValue - avgSound;
  micValue = System.Math.Abs(micValue);
  filter.Step((micValue / avgSound));

  intval = (int)(filter.SmoothedValue * 120);
  if (val> 100) { val = 100; }
  if (val< 8) { val = 0; }

  led.SetDutyCycle((uint)val);
  Thread.Sleep(5);

}
```

   After the sound reading, we work out how far it is above or below the average, and then make it positive with the `Absolute` method (we do this because whether it is 100 below or above the average, it means that it is 100-worth of sound). We then run the `Step` method of the filter with a value that represents the sound as a number between `0` and `1`. To set the brightness of the LEDs we need a percent (between `0` and `100`) so we need to multiply the current **SmoothedValue** by 100, but I found that the microphone didn't often get close to reporting full sound, so by multiplying the smoothed value by 120 we can get full brightness without bursting our eardrums. Feel free to play with that value until you get the sensitivity you need. Just keep in mind that setting it too high could make the LEDs light up even in silence. If the LEDs are lighting up in silence then an easy fix is to set the value to `0` if it is below a certain threshold. In the code above, the LEDs will not turn on until the microphone is reporting 8 percent or more sound. This does mean that when they do turn on, they will already be 8 percent bright, but that is low enough to not be jarring when behind the TV. Finally, we set the brightness of the LEDs, and pause for `5ms` before repeating.

8. Click on **Start** and try out your program. You can test it by blowing on the mic – the harder you blow, the brighter the LEDs should go. If they seem to flicker or change too quickly, decrease the smoothing factor (we used `0.03` to start with) until it seems right. On the other hand, if they seem sluggish, then increase the factor.

Not working?

Did you receive values back in step 7? If so, your microphone is working fine, but something has gone wrong with the way you have connected your LEDs, or you have not copied the code correctly. For both instances, double check the code, and follow the wiring exactly – it's very easy to put a wire in the wrong hole. Also, remember that although resistors do not have a right and wrong way around, almost everything else does.

If you still cannot get it to work, remove all the wires and components and put them back one by one, testing at each stage. So start by just plugging the microphone in with nothing else. Once that works, move onto the rest, and try to get a single LED to work. Once a single one is working then add the others.

Remember that the number of LEDs will also determine what resistors you need and whether to lay them out in series or parallel. So use the LED calculator each time you change the LEDs.

Other ideas and hints

- Using two more transistors you could wire up RGB LEDs and make the LEDs randomly change color or even make the color dependent on the sound level – so red could be when it's very loud, but a calm blue could be for when it is soft.
- Add a **Color** sensor to detect the ambient color, and make the RGB LEDs match. So, you could position the sensor just in front of the TV, and make the LEDs change to the same color as what is on-screen.
- Wondering how to power your Netduino when it isn't plugged into your PC? Steal your Smartphone wall charger (which will probably use the same USB cable as the Netduino), and power it from there! Avoid using Chinese non-branded chargers as they are potentially very dangerous to your hardware.
- Another handy way to power the Netduino (for this project) is to plug it directly into the TV's USB ports (most new TVs have these).

Summary

Well, if you made it this far successfully, then the good news is that you've gained the basic knowledge to complete everything in this book! You learned how to use a breadboard, how to read from a sensor with an Analog pin, and how to power things with a separate power supply while still controlling them straight from the Netduino.

The cool thing about electronics is that lots of the basic principles you've learned so far apply to a whole range of other projects. For instance, you could swap the microphone with a photocell sensor and control the LEDs based on how dark or light the TV screen is (note that you would need to change some code for this).

Next up we will add Bluetooth to our projects to control the Netduino from a phone.

3
Get Connected – Bluetooth Basics

Netduinos are great, and it is useful being able to attach a button or sensor to make it do something, but what if you want to control it with another device? What if you want to use your phone to turn LEDs on and off, or even drive a buggy around from across the living room?

Well, that's where Bluetooth comes in! In this chapter we will cover:

- How to connect a Bluetooth module
- Sending and receiving messages via Bluetooth
- Skills that can be applied to a variety of projects, and will be used in a number of other chapters where it makes sense to control the project with Bluetooth

Why Bluetooth?

There are other forms of wireless communication that we could use, like infrared and Wi-Fi, but Bluetooth is perfect for many household projects. It is cheap, very easy to set up, will typically use less power than Wi-Fi because of the shorter range, and it's very responsive.

It's important to keep in mind that there isn't a single "best" form of communication. Each type will suit each project (or perhaps budget) in different ways.

In terms of performance, I have found that a short message will be transmitted in under 20 milliseconds from one device to another, and the signal will work for just less than 10 meters (30 feet). These numbers, however, will vary based on your environment.

Things you need

The things required for this project are as follows:

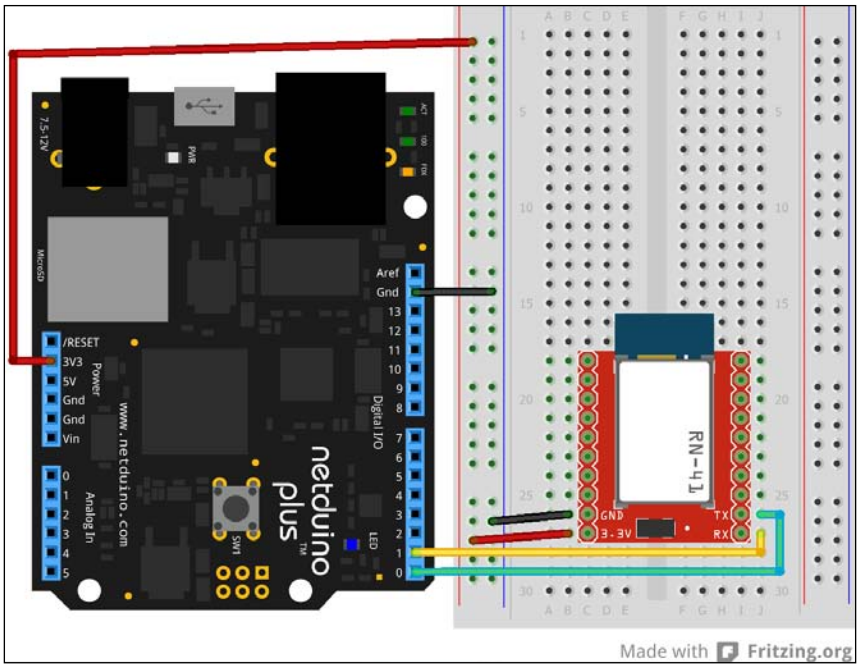
- Netduino
- Breadboard
- Bluetooth module
- Windows Phone 8

Lots of different Bluetooth modules exist, but I have found that the JY-MCU is very cheap (around $10) and reliable. Any Windows Phone 8 device can be used, as they all have Bluetooth.

The project setup

The setup for this project is extremely basic because we are just connecting the Bluetooth module and nothing else. Once our phone is connected we will use it to control the onboard LED, however, you can expand this to control anything else too. The Bluetooth module you buy may look slightly different to the diagram, but not to worry, just make sure you match up the labels on the Bluetooth module (GND, 3-3V or VCC, TX, and RX) to the diagram.

If you encounter a situation where everything is hooked up right but no data is flowing, examine the minimum baud rate in your Bluetooth module's manual or specifications sheet. It has been reported that some Bluetooth modules do not work well communicating at 9600 baud. This can be easily remedied by setting the baud rate in your SerialPort's constructor to 115200. For example, SerialPort(new SerialPort(SerialPorts.COM1, 115200, Parity.None, 8, StopBits.One).



Once it is wired up, we can get onto the coding. First we will do the Netduino part. The Netduino will listen for messages over Bluetooth, and will set the brightness of the onboard LED based on the percentage it receives. The Netduino will also listen for "ping", and if it receives this then it will send the same text back to the other device. We do this as an initial message to make sure that it gets from the phone to the Netduino, and then back to the phone successfully.

After that we will code the phone application. The phone will connect, send a "ping", and then wait until it receives the "ping" back. When the phone receives the "ping" back then it can start sending messages.

In this chapter only Windows Phone 8 will be covered, however, the same concepts apply, and it won't be too hard to code the equivalent app for another platform. The Netduino code will remain the same no matter what device you connect to.

Coding

Because we will be using a phone to connect to the Netduino, there are two distinct parts which need to be coded.

The Netduino code

1. Open up Visual Studio and create a new **Netduino Plus 2 Application**.
2. Just like in the previous chapters, add a reference to **SecretLabs.NETMF.Hardware.PWM**.
3. Open `Program.cs` from the **Solution Explorer**. You need to add the following `using` statements at the top:

```
using System.IO.Ports;
using System.Text;
using NPWM = SecretLabs.NETMF.Hardware.PWM;
```

4. You need to get the phone paired with the Bluetooth module on the Netduino. So in `Program.cs`, replace the `Main` method with this:

```
private static SerialPort _bt;
public static void Main()
{
    _bt = new SerialPort(SerialPorts.COM1, 9600,
    Parity.None, 8, StopBits.One);
    _bt.Open();
    while (true)
    {
        Thread.Sleep(Timeout.Infinite);
    }
}
```

   This code creates a new instance of a **SerialPort** (the Bluetooth module), then opens it, and finally has a loop (which will just pause forever).

5. Plug in your Netduino and run the code. Give it a few seconds until the blue light goes off—at this point the Bluetooth module should have a flashing red LED. On your Windows Phone, go to **Settings** | **Bluetooth** and make sure that it is turned on. In the list of devices there should be one which is the Bluetooth module (mine is called "linvor") so tap it to connect. If it asks for a pin try the default of "1234", or check the data sheet. As it connects, the red LED on the Bluetooth module will go solid, meaning that it is connected. It will automatically disconnect in 10 seconds; that's fine.
6. Now that you've checked that it connects correctly, start adding in the real code:

```
private static SerialPort _bt;
private static NPWM _led;
private static string _buffer;
public static void Main()
{
```

```
        _led = new NPWM(Pins.ONBOARD_LED);
        _bt = new SerialPort(SerialPorts.COM1, 9600,
        Parity.None, 8,   StopBits.One);
        _bt.DataReceived += new SerialDataReceivedEventHandler
        (rec_DataReceived);
        _bt.Open();
        while (true)
        {
            Thread.Sleep(Timeout.Infinite);
        }
    }
```

This is close to the code you replaced but also creates an instance of the onboard LED, and declares a string to use as a buffer for the received data.

7. Next you need to create the event handler that will be fired when data is received. Something that can easily trip you up is thinking that each message will come through as a whole. That's incorrect. So if you send a "ping" from your phone, it will usually come through in two separate messages of "p" and "ing". The simplest way to work around that is to just have a delimiter that marks the end of a message (in the same way as military personnel end radio communications by saying "10-4"). So send the message as "ping|" with a pipe at the end. This code for the `DataReceived` event handler builds up a buffer until it finds a pipe (|), then processes the message, then resets the buffer (or sets it to whatever is after the pipe, which will be the first part of the next message):

```
    private static void rec_DataReceived(object sender,
                        SerialDataReceivedEventArgs e)
    {
        byte[] bytes = new byte[_bt.BytesToRead];
        _bt.Read(bytes, 0, bytes.Length);

        char[] converted = new char[bytes.Length];
        for (int b = 0; b < bytes.Length; b++)
        {
            converted[b] = (char)bytes[b];
        }

        string str = new String(converted);
        if (str != null && str.Length > 0)
        {
            if (str.IndexOf("|") > -1)
            {
                _buffer += str.Substring(0, str.IndexOf("|"));
```

```
                ProcessReceivedString(_buffer);
                _buffer = str.Substring(str.LastIndexOf("|") +
                          1);
            }
            else
            {
                _buffer += str;
            }
        }
    }
```

At the start of the event handler, you create a byte array to hold the received data, then loop through that array and convert each byte to a char and put those chars into a char array. Once you have a char array, create a new string using the char array as a parameter, which will give the string representation of the array. After checking that it is not null or empty you check whether it has a pipe (meaning it contains the end of a message). If so, add all the characters up to the pipe onto the buffer and then process the buffer. If there is no pipe then just add to the buffer.

8. The only thing that remains is the method to process the received string (the buffer) and a method to send messages back to the phone. So put these methods below the event handler that you just added:

```
private static void ProcessReceivedString(string _buffer)
{
    if (_buffer == "ping")
    {
        Write(_buffer);
    }
    else
    {
        uint val = UInt32.Parse(_buffer);
        _led.SetDutyCycle(val);
    }
}

private static void Write(string message)
{
    byte[] bytes = Encoding.UTF8.GetBytes(message + "|");
    _bt.Write(bytes, 0, bytes.Length);
}
```

As mentioned before, if you receive a "ping" then just send it back, or alternatively convert the string into an unsigned integer and set the brightness of the onboard LED. When using Bluetooth in some of the next chapters, all the code that we have done will remain the same except for this process method, because each project will need to do something different with the messages.

The last method simply adds a pipe to the end of the string, converts it to a byte array, then writes it to the Bluetooth **SerialPort** to send to the phone.

At this point, you should run the code on the Netduino, but keep in mind that the same thing as before will happen because we are not sending it any data yet.

So next up, we need to make the phone application that helps us send messages to the Netduino.

The phone code

As mentioned, we will be using a Windows Phone 8 device to connect to the Netduino. The same principles demonstrated in this section will apply to any platform, and it all revolves around just knowing how to read and write the Bluetooth data. You may notice that much of the phone code resembles the Netduino code—this is because both are merely sending and receiving messages.

Before moving on, you will need the Windows Phone 8 SDK installed. Download and install it from here: `http://developer.windowsphone.com`

You may need to close any copies of Visual Studio that are open. Once it is installed you can go ahead and open the Netduino project (from the previous section) again, then follow these steps:

1. We could create the phone project in the same solution as the Netduino project, but in terms of debugging, it's easier to have them in separate instances of Visual Studio. So open up another copy of Visual Studio and click on **File** | **New** | **Project**. Find the **Windows Phone App** template by navigating to **Installed** | **Templates** | **Visual C#** | **Windows Phone**. Name the project and then click on **OK** to create it. A dialog may appear asking you to choose which version of the OS you would like to target. Make sure that Windows Phone OS 8.0 is selected (Windows Phone 7.1 does not have the required APIs for third party developers).

2. When creating a new Windows Phone application, `MainPage.xaml` will automatically be displayed. This is the first page of the app that the user will see when they run your app. XAML is the layout language used on Windows Phone, and if you've ever used HTML then you will be quite at home. In the **XAML** window, scroll down until you find the grid named `ContentPanel`. Replace it with:

```
<Grid x:Name="ContentPanel" Grid.Row="1"
  Margin="12,0,12,0">
    <Slider IsEnabled="False" Minimum="0" Maximum="100"
      x:Name="slider" ValueChanged="slider_ValueChanged"/>
</Grid>
```

   This will add a `Slider` control to the page with the value at the far left being `0` and the far right being `100`—essentially a percent. Whenever the user drags the slider, it will fire the `ValueChanged` event handler, which you will add soon.

3. That is the only UI change you need to make. So in the **Solution Explorer**, right-click on **MainPage.xaml | View Code**. Add these `Using` statements to the top:

```
using Windows.Storage.Streams;
using System.Text;
using Windows.Networking.Sockets;
using Windows.Networking.Proximity;
using System.Runtime.InteropServices.WindowsRuntime;
```

   We need to declare some variables, so replace the `MainPage` constructor with this:

```
StreamSocket _socket;
string _receivedBuffer = "";
bool _isConnected = false;
public MainPage()
{
    InitializeComponent();
    TryConnect();
}

private void slider_ValueChanged(object sender,
   RoutedPropertyChangedEventArgs<double> e)
{
    if (_isConnected)
    {
        Write(((int)slider.Value).ToString());
    }
```

```
}

async private void Write(string str)
{
    var dataBuffer = GetBufferFromByteArray(Encoding.UTF8.
        GetBytes(str + "|"));
    await _socket.OutputStream.WriteAsync(dataBuffer);
}

private IBuffer GetBufferFromByteArray(byte[] package)
{
    using (DataWriter dw = new DataWriter())
    {
        dw.WriteBytes(package);
        return dw.DetachBuffer();
    }
}
```

The StreamSocket is essentially a way to interact with the phone's Bluetooth chip, which will be used in multiple methods in the app. When the slider's value changes, we check that the phone is connected to the Netduino, and then use the `Write` method to send the value. The `Write` method is similar to the one we made on the Netduino, except it requires a few lines extra to convert the byte array into an `IBuffer`.

4. In the previous step, you might have noticed that we ran a method called `TryConnect` in the `MainPage` constructor. As you may have guessed, in this method we will try to connect to the Netduino. Add this method below the ones you added previously:

```
private async void TryConnect()
{
    PeerFinder.AlternateIdentities["Bluetooth:Paired"] =
    "";
    var pairedDevices = await PeerFinder.
        FindAllPeersAsync();

    if (pairedDevices.Count == 0)
    {
        MessageBox.Show("Make sure you pair the device
                        first.");
    }
    else
    {
```

```
            SystemTray.SetProgressIndicator(this,
              new ProgressIndicator { IsIndeterminate = true,
              Text = "Connecting", IsVisible = true });
            PeerInformation selectedDevice = pairedDevices[0];
            _socket = new StreamSocket();
            await _socket.ConnectAsync(selectedDevice.
              HostName, "1");
            WaitForData(_socket);
            Write("ping");
        }
    }
```

We first get a list of all devices that have been paired with the phone (even if they are not currently connected), and display an error message if there are no devices. If it does find one or more devices, then we display a progress bar at the top of the screen (in the **SystemTray**) and proceed to connect to the first Bluetooth device in the list. It is important to note that in the example code we are connecting to the first device in the list—in a real-world app, you would display the list to the user and let them decide which is the right device. After connecting, we call a method to wait for data to be received (this will happen in the background and will not block the rest of the code), and then write the initial ping message.

5. Don't worry, we are almost there! The second last method you need to add is the one that will wait for the data to be received. It is an asynchronous method, which means that it can have a line within it that blocks execution (for instance, in the following code the line that waits for data will block the thread), but the rest of the app will carry on fine. Add in this method:

```
async private void WaitForData(StreamSocket socket)
{
    try
    {
        byte[] bytes = new byte[5];
        await socket.InputStream.ReadAsync(bytes.
          AsBuffer(), 5, InputStreamOptions.Partial);
        bytes = bytes.TakeWhile((v, index) =>
         bytes.Skip(index).Any(w => w != 0x00)).ToArray();
        string str = Encoding.UTF8.GetString(bytes, 0,
         bytes.Length);
        if (str.Contains("|"))
        {
            _receivedBuffer += str.Substring(0,
              str.IndexOf("|"));
            DoSomethingWithReceivedString(_
              receivedBuffer);
```

```
                    _receivedBuffer = str.Substring(str.
                    LastIndexOf("|") + 1);
                }
                else
                {
                    _receivedBuffer += str;
                }
            }
            catch
            {
                MessageBox.Show("There was a problem");
            }
            finally
            {
                WaitForData(socket);
            }
        }
```

Yes, this code looks complicated, but it is simple enough to understand. First we create a new byte array (the size of the array isn't too important, and you can change it to suit your application), then wait for data to come from the Netduino. Once it does, we copy all non-null bytes to our array, then convert the array to a string. From here, it is exactly like the Netduino code.

6. The final code left to write is the part that handles the received messages. In this simple app, we don't need to check for anything except the return of the "ping". Once we receive that ping, we know it has connected successfully and we enable the slider control to let the user start using it:

```
private void DoSomethingWithReceivedString(string buffer)
{
    if (buffer == "ping")
    {
        _isConnected = true;
        slider.IsEnabled = true;
        SystemTray.SetProgressIndicator(this, null);
    }
}
```

We also set the progress bar to null to hide it.

7. Windows Phone (and other platforms) needs to explicitly define what capabilities they require for security reasons. Using Bluetooth is one such capability, so to define that we are using it, in the **Solution Explorer** find the **Properties** item below the project name. Left-click on the little arrow on the left of it to expand its children. Now double-click on **WMAppManifest.xml** to open it up then click the **Capabilities** tab near the top. The list on the left defines each specific capability. Ensure that both **ID_CAP_PROXIMITY** and **ID_CAP_NETWORKING** are checked.

And that's it! Make sure your Netduino is plugged in (and running the program you made in this chapter), then plug your Windows Phone 8 in, and run the code. The **run** button may say **Emulator X**, you will need to change it to **Device** by clicking on the little down arrow on the right of the button.

Once the two devices are connected, slide the slider on the phone forwards and backwards to see the onboard LED on the Netduino go brighter and dimmer.

Not working?

If the phone does not connect after a few seconds then something has probably gone wrong. After double-checking your wiring, the best thing to try is to unplug both the Netduino and phone, then plug them back in. If you are using a different Bluetooth board, you may have to pair it again to the phone. Repeat step 5 of the *The Netduino Code* section of this chapter. With both plugged back in, run the Netduino code (and give it a few seconds to boot up), then run the phone code. If that still doesn't work, unplug both again, and only plug back in the Netduino. When it is powered up, it will run the last application deployed to it. Then with your phone unplugged, go to the app list and find the phone app you made, and tap on it to run it.

Summary

You've managed to control your Netduino from afar!

This chapter had a lot more code than most of the rest will because of needing to code both the Netduino and phone. However, the knowledge you've gained here will help you in many other projects, and we will be using this chapter as a base for some of the others.

In the next chapter, we will learn how to use a microphone to turn a lamp on and off, which is actually the cornerstone to controlling devices with a Netduino.

4
Let There Be Light – By Clapping or Tapping

If you've ever watched a cheesy 80's romance movie, you will know what a clapper is. Hop into bed and clap twice: The lights dim, a disco ball lowers from the ceiling, and some Barry White starts playing in the background. Thankfully, the 80's are gone, so we will not be making that. We will, however, be switching a light `ON`/`OFF` with clapping.

In this chapter we will cover:

- Detecting peaks in sound to react when someone claps
- Using a mechanical relay to switch high-voltage electricity
- Protecting your circuit with a **flyback** diode

Double clap

Most clapper systems are generally quite dumb. This means that to determine whether the user is clapping, they just analyze the volume of noise and see if it goes past a threshold. This means that any noise will trigger the system, not just a clap. A simple fix for this is to require two claps (or loud noises) successively, which will remove a lot of false positives. Of course, this is a very basic approach and you can spend a great deal of time tweaking the code to reduce the number of false positives.

Déjà vu…

This is going to be a rather easy project, because most of it is very much like the one in *Chapter 2*, *Lights, Camera, Action – Sound-Controlled Ambient LEDs*. In fact, they are so similar that before moving on, you should have that setup (you can create a copy of the Visual Studio project) and code because we will work on top of that.

Dealing with electricity is very dangerous. The text below is a general guide, and does not waive your right to use common sense. If you do not have experience with high-voltage electricity, then it's safer to find a friend who does, that can help you out. Do not attempt this chapter while in the bath.

Things you need

- Netduino
- Breadboard
- Wall plug
- Lamp
- TIP122 transistor (or equivalent NPN transistor)
- Electret microphone (with breakout board)
- 1K ohm resistor
- Diode Rectifier (1N4001)
- SPDT Relay (Relay SPDT Sealed)

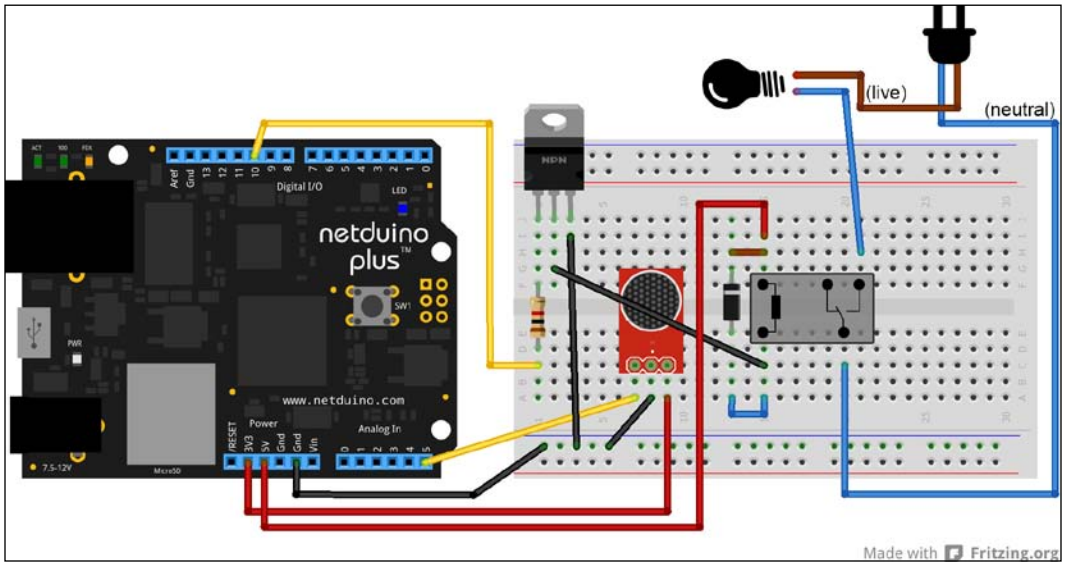
A wide range of relays exist, and may work fine as a replacement with little to no changes.

The project setup

Take note of the differences to *Chapter 2*, *Lights, Camera, Action – Sound-controlled Ambient LEDs*, before plugging things into or out of your breadboard. In this chapter, instead of switching a few small LEDs `ON` and `OFF`, we will be using a relay to switch a high voltage (up to 240v mains) lamp `ON` and `OFF`.

A relay is similar, in a way, to the transistor that you were using before, but this particular item is a mechanical relay which physically connects and disconnects wires inside its box (you even get to hear a nice "click" when it switches!). It has five pins, which might not be marked on the unit, so make sure you read its datasheet. The negative and positive pins determine whether the switch is `ON` or `OFF`. When power (5V in this case) is connected to those pins, then the switch is turned `ON` (connected), otherwise it is `OFF` (disconnected). The remaining three are used to connect the wire to your actual light and wall socket.

There are two states – `OFF`, which is marked as **NC** (**Normally Closed**) and can be used if you only want power when the relay is not powered; and `ON`, marked as **NO** (**Normally Open**) which will be used when you want to power the relay to activate the switch. We will be ignoring the NC pin as we only want our light to go on when we turn the relay on.



In *Chapter 2*, *Lights, Camera, Action – Sound-controlled Ambient LEDs*, we used a separate battery pack to power the LEDs. This time, all the electronics are powered directly from the Netduino, and the main lamp electricity never enters the same circuit (it merely goes through the mechanical relay (think of it as a physical switch). We use the transistor as a way to connect and disconnect power to the relay (yes, basically you are using a mini switch to turn `ON`/`OFF` a big switch), which in turn allows the mains' power to go through the lamp. From the wall plug you should have two or three wires (note that colors vary, and could even be wired wrong in the plug): **Neutral** (blue), **Live** (brown/red), and **Earth** (sometimes). The Live and Earth (if it has) from the plug will go directly to the lamp as usual. The Neutral wire however will go from the plug into the relay and then out the other side of the relay (on the NO pin) to the lamp. Before connecting the mains up to your project, I recommend that you test with your battery pack from *Chapter 2*, *Lights, Camera, Action – Sound-controlled Ambient LEDs* (and light up some LEDs).

Although only lamps have been mentioned, virtually any appliance can be powered like this. But keep in mind that the relay has a specified maximum Amperage, which you need to stay below. The relay we are using here is `5A` and will power most lamps fine.

The diode is used as a flyback diode (to stop voltage spikes when switching the relay).

In a real-world scenario, an opto-isolator/opto-coupler would be used too, but it is not important right now.

The Netduino code

The code might look a bit long and confusing to start, but once you read over it, things will become clearer. You will almost definitely want to adjust the code for your particular scenario. As mentioned before, create a copy of the project you made in *Chapter 2*, *Lights, Camera, Action – Sound-controlled Ambient LEDs*; we will use that as the base:

1. In `Program.cs` delete the line that declares the `led` variable. Although we will be using the same pin, we no longer want to use PWM because the relay will either be `ON` or `OFF` (you cannot fade a switch). So in its place we declare an `OutputPort` function which is similar to a PWM pin except that it is only `ON` or `OFF`.

   ```
   OutputPort relay = new OutputPort(Pins.
     GPIO_PIN_D10, false);
   ```

2. We want the microphone values to change faster, and be less smoothed out, so change the value that you sent to the `LowPassFilter` constructor to `0.2`.

3. Just above the `while` loop, we need to declare some variables. `silenceValue` is a setting which determines what noise level is considered silent (between the two claps we need silence). And `timeoutValue` specifies how many times the code can loop between claps before resetting to listening for an initial clap:

   ```
   intsilenceValue = 20;
   inttimeoutValue = 100;
   intloopsSinceFirstClap = 0;
   boolisWaitingForSecondClap = false;
   boolisWaitingForSilence = false;
   boolisActivated = false;
   ```

4. Within the `while` loop, leave the existing code, except for the `if` statements and code below them. The easiest way to understand the code is to study it, but in summary. Each loop, we check whether the sound is above the threshold (which means this is a clap), if so we set a flag to ensure that the sound drops below the silence threshold before listening for the next clap. Once it becomes silent, we send another flag specifying that only a single clap is now needed. If that clap comes before the timeout is reached then we switch the relay. To avoid any confusion, the whole `while` loop is below:

```
while (true) {
  doublemicValue = mic.Read();
  micValue = micValue - avgSound;
  micValue = System.Math.Abs(micValue);
  filter.Step((micValue / avgSound));
  intval = (int)(filter.SmoothedValue * 120);

  loopsSinceFirstClap++;
  if (isWaitingForSilence&&val<silenceValue) {
    isWaitingForSilence = false;
    isWaitingForSecondClap = true;
    Debug.Print("Reached silence after " +
      loopsSinceFirstClap);
  }

  if (isWaitingForSecondClap&&loopsSinceFirstClap>
    timeoutValue) {
      isWaitingForSecondClap = false;
      Debug.Print("Timeout");
  }

  if (val>silenceValue) {
    if (isWaitingForSecondClap) {
      isWaitingForSecondClap = false;
      isWaitingForSilence = true;
      isActivated = !isActivated;
      relay.Write(isActivated);
      Thread.Sleep(500);
      Debug.Print("Second clap");
    }
```

```
      else if (!isWaitingForSilence) {
        isWaitingForSilence = true;
        loopsSinceFirstClap = 0;
        Debug.Print("First clap");
      }
    }
    Thread.Sleep(5);
}
```

Once the relay is switched, we sleep for half a second to stop the light switching again due to extra noise. There are a few lines that output values to the **Output** windows in Visual Studio (when debugging) – these are to help you with testing and seeing what is going on.

5. Running the code now should have the relay happily clicking away as you clap. Tapping the microphone can be easier than actually clapping when you are just testing.

Other ideas and hints

- The wires from a wall plug will not go into a breadboard so you can solder them directly to the relay pins, or go down to your local hardware store and see what they can provide you with.
- Create a smarter clapper which can detect more complicated patterns than a double-clap. Something like this could be used on a door to react when someone knocks.
- Using your Bluetooth skills, connect up your phone to switch the light `ON` and `OFF`.
- Try to power some other devices besides a lamp (keeping in mind the limits of your relay). Take out the microphone where not applicable.

Summary

You learned how to connect your code with the physical world by powering other devices. That is no small feat, and is one of the easiest, most fulfilling things you will learn to do in the wonderful world of electronics. With this knowledge you can do everything from remotely turning a kettle `ON`, to opening a garage door from your phone.

In the next chapter we will be controlling a garage door or gate with our minds (well, with a Smartphone at the very least)!

5
Honey, I'm Home – Automated Garage Doors with Your Mobile Phone

What do you treasure most in life, the one thing that will never leave your side? Yup, your phone, which makes it a perfect candidate for using as a remote for your gate or garage door. It also means you can give friends or relatives "spare keys" without having to buy more remotes!

In this chapter we will cover:

- What a normal push button is and how to use it
- Using a relay to fake the press of a button
- Modifying the Bluetooth code to use it for other projects

Various approaches

There are a couple of different ways that you could achieve opening a garage door with the Netduino. The first, but most difficult, is to make an actual remote. Along with the electronics being very complicated, it will also be hard to set it to the right signal. Another problem is that there isn't a single standard remote, and you would need to work out how to make the variation that works with your garage door. Another difficult way is to wire your Netduino directly up to the circuit board that controls the garage door—this is both complicated, and will vary drastically based on your situation. The next way, which is how we will be doing it, is to use an existing remote that already works with your garage door or gate, and use the Netduino to simply *fake* a press of the button.

Things you need

The things required for this project are as follows:

- Netduino
- Working remote for your gate/garage door
- Soldering iron
- Breadboard
- TIP122 transistor (or equivalent NPN transistor)
- 1K ohm resistor
- 10K ohm resistor
- Diode Rectifier (1N4001)
- SPDT Relay (Relay SPDT Sealed)

Modifying your remote

Grab a remote that currently works, and pop it open—you may need to remove a screw or two, but you're basically an electronic engineer by now and will manage just fine. Once opened, you need to find the button that is soldered onto the board. Most buttons will have four pins, and look something similar to this:

The principles of a four-pin button/switch are simple. When the button is pressed, the pins diagonal to each other are connected together. So to *fake* the press of the button, you can touch a wire between the top-left and bottom-right (or the top-right and bottom-left pins) pins. You can see this in action on your Netduino by plugging it in, waiting for the light to go off, and connecting the diagonal pins of the Netduino's onboard switch with a wire. As soon as you short out the Netduino's switch, the blue LED will light up, which means that it is restarting. This is what happens when you physically press the button.

Now get the remote you opened up and test the same theory on it by connecting a wire to diagonal pins. When you do this the remote light should turn on and the garage door should be opened/closed.

When using a wire to do this you should press the wire firmly into the pin and try not to shake, as it will be equivalent to repeatedly pressing the button and may confuse its little brain.

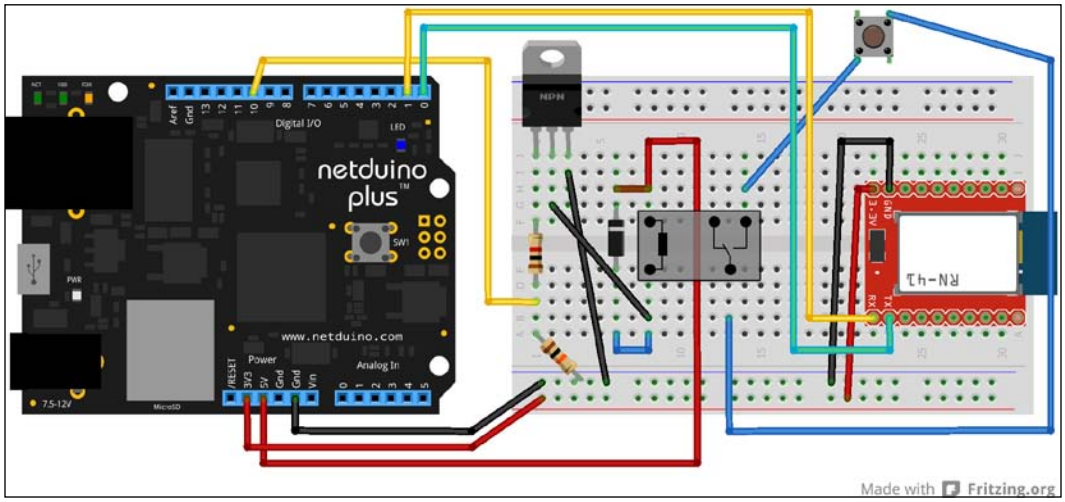
Once you are confident that you know which two pins activate the remote, you can start heating up your soldering iron!

Remove the battery from the remote in case you short anything. Take a wire and solder it to the first pin, then take another wire and solder it to the other pin. Be very careful to not add too much solder, as it could drip onto neighboring circuitry and cause a short. Once the two wires are soldered onto the remote, put that battery back in and tap the unsoldered ends of the two wires together to connect them. If you have soldered correctly, then the remote should open/close the garage door. The cool thing about this method is that the button will also still work fine by just pressing it.

Setting up the remote

Hopefully, you still have your breadboard setup from the previous chapter, because this one is almost exactly the same! To fake the press of the button, we need to connect those two wires you soldered onto the remote, which is exactly what we did when we connected the wires going to the lamp earlier.

For this setup, however, we do not need the microphone, and we do need a Bluetooth module (wired as before). You will also notice in the following diagram that there is now a 10K resistor joining the Digital 10 pin and GND. This is called a **pull-down resistor**, and will stop the relay from immediately switching when the Netduino powers up. If we don't add this, when the power goes off and on, the Netduino will open your garage door automatically—not the greatest security system.



Coding

We are going to be controlling the Netduino from a Windows Phone, so we have to execute two separate code projects.

Netduino code

In *Chapter 3, Get Connected – Bluetooth Basics*, we connected a phone to the Netduino over Bluetooth. That chapter was the base for all others that use Bluetooth, so find the Netduino project you created, make a copy of it, and follow these steps:

1. Open the copy that you just made and go to `Program.cs`. Remove the LED variable, along with the lines of code that use it.
2. We now need to declare the variable for the Digital 10 pin so that we can turn the relay on and off. So, after the other variables, add this line:

   ```
   private static OutputPort _relay;
   ```

   And then initialize it within the `Main` method:

   ```
   _relay = new OutputPort(Pins.GPIO_PIN_D10, false);
   ```

3. Scroll down to the `ProcessReceivedString` method and replace it with the following code snippet:

```
private static void ProcessReceivedString(string _buffer)
{
    switch (_buffer)
    {
        case "ping":
            Write(_buffer);
            break;
        case "switch":
            _relay.Write(true);
            Thread.Sleep(500);
            _relay.Write(false);
            break;
    }
}
```

   All that those lines do is check if the message is a ping or a `switch` command. If it is a `switch` command then it should turn the relay on, wait half a second, and then turn it off. This will be equivalent to pressing the remote button for half a second.

That is everything for the Netduino, I told you it was similar to the previous projects!

The phone code

Just like the Netduino part, create a copy of the Bluetooth Windows Phone project from *Chapter 3*, *Get Connected – Bluetooth Basics*.

1. Open the project and bring up `MainPage.xaml`. Currently, there is a Slider control present. Delete that line of XAML and replace it with the following line of code which will create a button:

```
            <Button x:Name="switchBtn" Content="Open/Close"
Click="switchBtn_Click" IsEnabled="False"></Button>
```

2. Next open the `MainPage.xaml.cs` code file. Because we deleted the Slider control, any code that references it needs to be changed. So delete the `slider_ValueChanged` method completely.
3. When the ping came back, the Slider was enabled. So in the `DoSomethingWithReceivedString` method, change `slider.IsEnabled = true;` to `switchBtn.IsEnabled = true;`.

4. The final change is to make the button send the `switch` message. Below your other methods, create the event handler for the button `click`:

```
private void switchBtn_Click(object sender,
RoutedEventArgs e)
{
    Write("switch");
}
```

Those are all the changes needed. So you can now run the Netduino code, then run the Windows Phone code. After connecting, you can tap the button to open and close the garage door!

Other ideas and hints

Here are a few ideas and hints that you can consider while creating your project:

- Add a sensor (possibly a flex sensor) to determine whether the gate is open or closed, and report the result back to the phone.
- Currently, this is *very* insecure. If someone knows that "switch" will open it, then they could easily get it open. So change that keyword to something more complicated.
- Allow the garage or driveway lights to be turned on when the garage door/gate opens.

Summary

By now you should have realized that the simple act of using a relay can open a whole world of new possibilities! We used the same circuit to control a lamp as we did to open a garage door. You also made very minor changes to the base Bluetooth projects to completely change their purpose.

In the next chapter, we will raise a flag when an e-mail is received, and even display the subject line on an LCD.

6

You've Got Mail, and Here's a Flag to Prove It

Manipulating physical objects via code is one of those things that gives even the manliest man a warm fuzzy feeling in his stomach. An example of this was the previous project where we opened a garage door from a smartphone. Now, we can take this feeling even further by making the manipulation happen based on someone else's actions, for example, when someone sends us an e-mail.

In this chapter we will cover:

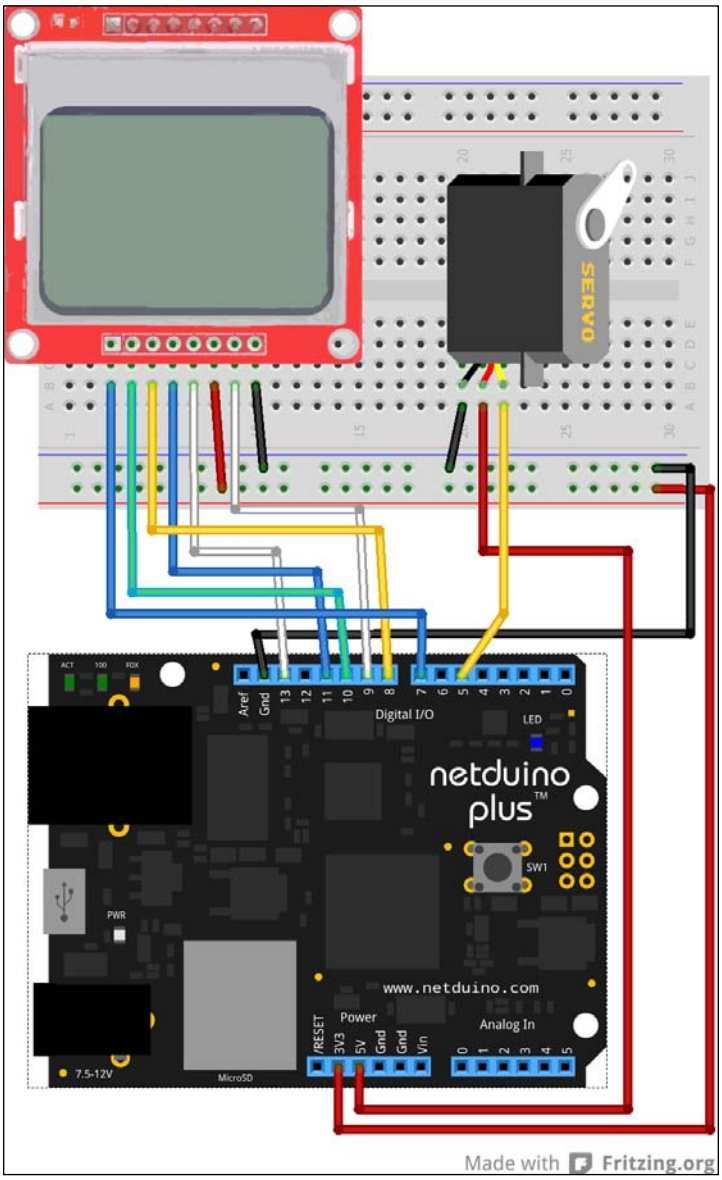
- Creating our first internet-enabled application
- Connecting to an e-mail service with POP3 and checking for mail
- Using a Nokia 5110 LCD
- Controlling a servo

Things you need

- Netduino (must be a Plus or Plus 2 for the Ethernet port)
- Internet and Ethernet cable to connect the Netduino
- Breadboard
- Nokia 5110 LCD
- Micro servo

The project setup

In this project we are using a Nokia 5110 LCD screen to display part of an e-mail. These displays are great because they are easy to use and really cheap (as low as $5). Something to watch out for when connecting it up though is that different versions will have different pin layouts. So use the image below as a reference, but read the text written next to each pin to be sure.



Here are the pin mappings:

Nokia 5110 LCD	Netduino Plus
VCC/3V3	3V3 Pin
GND	GND Pin
ChipSelect/CE	Pin 10
Reset/RST	Pin 7
DataCommand/DC	Pin 8
MOSI/DIN	Pin 11
SCLK/Clk	Pin 13
Backlight/BL	Pin 9

The servo is connected to the Netduino platform's 5V power pin and GND to power it. The signal wire (usually orange or yellow) is connected to Digital Pin 5.

With the servo plugged in, get it secured on a platform and strap/stick a nice little flag with a stick onto the rotating end.

The Netduino code

As always, the first thing to do is to create a new Netduino Plus 2 project (or Netduino Plus project, if that is what you are using). With that done, let's add some third-party libraries!

Libraries are your friends

One of the great things about having a big community around the Netduino is that many libraries exist for common tasks. We will be using a few for this project. Follow the instructions to get each one.

.NET Micro Framework Toolbox

The **.NET Micro Framework Toolbox** subframework was written by Stefan Thoolen and contains the code to control a wide range of components and provides helpers for many common tasks. We will be using it to help us connect to an e-mail account with POP3:

1. Head over to `http://netmftoolbox.codeplex.com/SourceControl/latest` and click on the **Download** tab on the upper-right side. This will download the latest version of all the binaries and the source code.

2. When the download is done, right-click on the compressed folder, select **Properties** and then click on **Unblock**. This step is necessary for any internet downloaded binaries; otherwise, the compiler will not build the project. If you don't see an **Unblock** option, it is already unlocked.
3. Now you can unzip the archive somewhere. It's good practice to keep your downloaded libraries and toolkits in a common `Netduino Helpers` folder. This makes for fast and easy referencing in future projects.
4. In Visual Studio add a reference to a DLL just like you have before, except this time select the **Browse** tab at the top then navigate to the folder containing the files you just unzipped. Add **Toolbox.NETMF.NET.Core**, **Toolbox.NETMF.NET.POP3_Client**, and **Toolbox.NETMF.NET.Integrated** from the **Release (4.3)** folder.
5. Now add the following line of code to the top of `Program.cs`:

```
using Toolbox.NETMF.NET;
```

Nokia 5110 LCD

Another Netduino community member, Omar, created a library to easily display things on the Nokia 5110 LCD. We need to make a small change because this was written for a previous version of the firmware:

1. Download the **Nokia 5100 LCD.zip** file from `http://wiki.netduino.com/Nokia-5110-LCD.ashx` and extract it
2. Copy the `Nokia.cs` file and paste it into the project by right-clicking on **Solution Explorer** and then **Paste**.
3. Reference the **SecretLabs.NETMF.Hardware.NetduinoPlus** and **SecretLabs.NETMF.Hardware.PWM** DLLs.
4. Add the following line to the top of the `Nokia.cs` file:

```
using NPWM = SecretLabs.NETMF.Hardware.PWM;
```

5. Scroll to line **114** and replace the line of code with the following:

```
private NPWM backlight = null;
```

   If you don't see line numbers in Visual Studio, turn them `ON` by navigating to **Tools | Options | Text Editor | All Languages**, and check the **Line Numbers** checkbox.

6. Go down a few lines to **122** and change the constructor header to:

```
public Nokia_5110(bool useBacklight, Cpu.Pin latch,
  Cpu.Pin backlight, Cpu.Pin reset, Cpu.Pin dataCommand)
```

7. The last change is to replace line **126** with the following:

```
this.backlight = new NPWM(backlight);
```

8. In `Program.cs` add the following to the top:

```
using Nokia.LCDScreens;
```

The Netduino Servo class

On the Netduino forums, Chris Seto has created a very easy-to-use class for controlling servos. Just like the `Nokia 5100 LCD` class, we need to change some things:

1. Open `http://forums.netduino.com/index.php?/topic/160-netduino-servo-class/page-3#entry36279` and copy the whole block of code.
2. Create a new class called `Servo.cs` in your project and delete any code that is automatically added to the file, then paste the `Servo` class.
3. Scroll to line **30** and replace it with:

```
private Microsoft.SPOT.Hardware.PWM servo;
```

4. Change line **49** to:

```
servo = new Microsoft.SPOT.Hardware.PWM(
  (Cpu.PWMChannel)channelPin, 20000, 1500,
    Microsoft.SPOT.Hardware.PWM.ScaleFactor.
      Microseconds, false);
```

5. Scroll down to line **51** and add the following:

```
servo.Start();
```

6. Add the following line of code to the top of `Program.cs`:

```
using Servo_API;
```

Receiving e-mail

After including all those libraries we can move on to creating our code:

1. In the `Main` method we initialize the LCD, Servo, and POP3 client:

```
POP3_Client Mailbox = new POP3_Client(new
  IntegratedSocket("pop.yourmail.com", 110), "username",
    "password");
Servo servo = new Servo(PWMChannels.PWM_PIN_D5);
Nokia_5110 Lcd = new Nokia_5110(true, Pins.GPIO_PIN_D10,
  Pins.GPIO_PIN_D9, Pins.GPIO_PIN_D7, Pins.GPIO_PIN_D8);
```

Obviously, you will need to replace the strings with the POP3 settings from your e-mail provider. Unfortunately, Netduino isn't powerful enough to handle SSL, which means that providers like Gmail (which requires SSL) will not work. An example of a free e-mail provider that does not require SSL is `http://inbox.com`.

2. Next we just have a constant loop checking for new e-mail (every 5 seconds). If a new e-mail is received, then the sender and subject will be displayed on the LCD. Additionally, the servo will rotate. After 5 seconds the servo will go back to its original position.

```
Lcd.BacklightBrightness = 100;
uint numberOfEmails = 0;
while (true) {
  servo.Degree = 20;
  Mailbox.Connect();
  if (Mailbox.MessageCount != numberOfEmails) {
    if (numberOfEmails != 0) {
      servo.Degree = 160;
      uint[] Id, Size;
      Mailbox.ListMails(out Id, out Size);
      string[] Headers = Mailbox.FetchHeaders(
        Id[Id.Length - 1], new string[] { "subject",
          "from" });
      Lcd.Clear();
      Lcd.WriteText(Headers[0] + "     -     " +
        Headers[1]);
    }
    numberOfEmails = Mailbox.MessageCount;
  }
  Mailbox.Close();
  Thread.Sleep(5000);
}
```

3. Running that will turn on the LCD backlight and move the servo to its starting point, but nothing more until a new e-mail arrives. Notice in the code, the first time we check for e-mail and receive the number of messages (total in inbox) we don't react to – this is so when you turn ON the Netduino, it doesn't automatically think there is a new e-mail.

Other ideas and hints

- If you have some web development skills you could actually get SSL POP3 services working by creating a sort of proxy to do all the work and send the messages down to the Netduino, unencrypted.
- Instead of lowering the flag after 5 seconds, you could override the default action of the on-board button to make it a clear switch. The flag could stay up until the button is pressed.
- Make the e-mail text scroll on the screen.
- Flash or pulse some LEDs or the LCD backlight when an e-mail comes in.
- Make annoying noises with a Piezo speaker.
- Notice how the power LED on the Netduino dims slightly when the servo moves? That is because servos draw lots of power. For anything more than a tiny servo you should power it from a separate battery pack. Run the battery pack to the servo as normal, but also connect the Netduino ground pin to the ground of the battery.

Summary

This was quite a "code-heavy" chapter, but had nothing too complicated; and you made your first internet connected program for the Netduino! You also learned the importance of the community, and what awesome resources it provides.

In the next chapter we will be having some juvenile fun, making a homemade breathalyzer!

7
I'm Completely Dude, Sober – a Homemade Breathalyzer

What is a good book without a section on drinking? Yep, this chapter is just pure juvenile fun.

Before we get started, let me be frank. Unfortunately, with cheap non-specialized components, a homemade breathalyzer will never be accurate. The first reason is that the sensors are not sensitive enough to get a precise value. Then there is the problem of it being close to impossible to accurately calibrate your system. A basic alcohol sensor will not return a simple **Blood Alcohol Content** (**BAC**) value, so you will need to do extensive testing comparing the values you get back with an industrial tester. Not only is it hard to find a cop willing to let you play with their toys, it will also only help to a certain extent. Environmental variables such as temperature and humidity also play a role, so you may get your sensor perfectly calibrated and then as soon as you leave your area it is completely wrong.

So what's the good news? Well, although we cannot get a specific BAC, we can absolutely use this to compare results between people. Yes, this chapter is about creating the world's stupidest bar game: "Who's the drunkest?"

Due to not being able to get an accurate BAC level, this breathalyzer will *never* be able to determine if you are over the legal driving limit. Regardless, if you need to use a breathalyzer to see if you are OK to drive, you probably shouldn't be driving at all.

In this chapter we will cover:

- What a voltage divider is and how to make one
- Connecting up a basic alcohol sensor

Things you need

Following is the hardware required to make the breathalyzer:

- Netduino
- Breadboard
- Nokia 5110 LCD
- MQ-3 alcohol sensor with breakout (the one from DealExtreme is the specific one used here)
- 100 ohm and 200 ohm resistor

Setting up the breathalyzer

A common problem when using sensors is that some are 5V. But why does this matter?

Voltage dividers

Although the Netduino is very similar to the Arduino, a major difference is that the Netduino uses 3.3V instead of 5V. That means that the pins can only read up to 3.3V (although they are 5V tolerant and will not explode when supplied with 5V), which is a bit of a problem because a lot of sensors run on 5V. Powering the sensors isn't a problem because there is a 5V pin on the Netduino, but any voltage returned that is above 3.3V will get lost. So the solution is to use a basic voltage divider which is designed to divide the voltage with just two resistors.

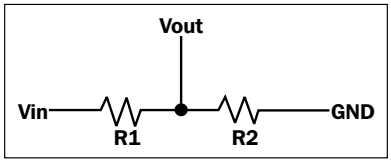
To determine the ohms of those resistors we use a basic formula:

$$\text{outputV} = \text{inputV} \times \left(\frac{\text{Resistor2}}{\text{Resistor1} + \text{Resistor2}} \right)$$

In this equation, the *input* voltage will be 5V (because the sensor is 5V), and we need the *output* voltage to be close to 3.3V. So we are going to use (although you can substitute your own numbers) a 100 ohm resistor for *Resistor1* and a 200 ohm resistor for *Resistor2* which will give us:

$$3.3 = 5 \times (200 / (100 + 200))$$

Once we know what the two resistors are it is very easy to wire up:

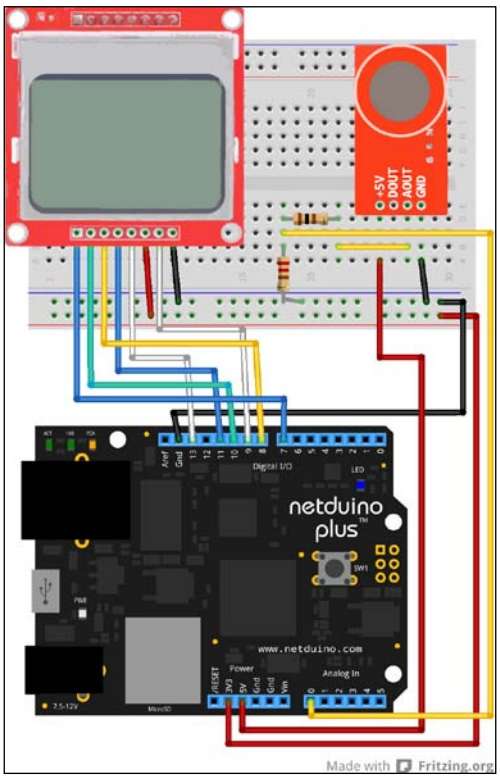


Now that we know how to reduce the voltage, let's move onto plugging everything in!

Hardware

Hopefully your LCD is still connected from the previous chapter, because we will be using it in exactly the same way here.

Next to it we are connecting the alcohol sensor. The sensor is on a board that has a block at the bottom and will be impossible to push into a breadboard. So just extend wires from the board pins to the breadboard. We are not using the DOUT pin, only +5V, AOUT, and GND.



The code

With the hardware side done, the code is pretty easy. The only new thing here is that last time we used an analog input pin, and we included the Netduino `AnalogInput` DLL. This time we will be using the built-in `AnalogInput` from the .NET Micro Framework. After using both, you won't get confused when reading other peoples code, regardless of which they use:

1. Create a new Netduino Plus 2 project.
2. Return to the previous chapter and include the `Nokia.cs` code just as before. You can copy the code from the previous project instead of redoing the modifications we had to do.
3. Within the `Main` method in `Program.cs`, we need to create new instances of the LCD and `AnalogInput` pin, and make the LCD backlight glow:

```
AnalogInput alcoholSensor = new AnalogInput(AnalogChannels.ANALOG_
PIN_A0);
Nokia_5110 Lcd = new Nokia_5110(true, Pins.GPIO_PIN_D10, Pins.
GPIO_PIN_D9, Pins.GPIO_PIN_D7, Pins.GPIO_PIN_D8);
Lcd.BacklightBrightness = 100;
```

4. This alcohol sensor has a heating element inside to help get readings, but it doesn't heat up instantly. So we need to give it some time to heat up. This wait time will also allow the sensor to return back to its base value after someone has blown on it. Include the following code that will wait 3 minutes while also letting the user know the progress:

```
for (int i = 0; i < 100; i++)
{
    Lcd.Clear();
    Lcd.WriteText("Warming up (" + i + "%)");
    Thread.Sleep(1800);
}
```

5. Once the sensor is heated up and is getting normal readings, we need to determine a baseline which will be the average value that it is getting with "clean" air over a few seconds:

```
double totalReading = 0;
for (int i = 0; i < 100; i++)
{
    Lcd.Clear();
    Lcd.Refresh();
    Lcd.WriteText("Calibrating (" + i + "%)");
    totalReading += alcoholSensor.Read();
    Thread.Sleep(100);
}
double baseLineReading = totalReading / 100;
```

6. Now that the sensor has warmed up, and is reading a baseline value, we need to start reading and displaying an actual value. As mentioned, we cannot get a proper BAC value, so in the code we instead display the new reading as a percent of the baseline. When you blow you may get **800%** displayed which will mean that the value is eight times higher than the baseline. We also record the max value:

```
int maxReading = 0;
while (true)
{
    Lcd.Clear();
    Lcd.Refresh();
    int difference = (int)((alcoholSensor.Read() /
      baseLineReading) * 100);
    if (difference > maxReading) { maxReading = (int)
      difference; }
    Lcd.WriteText(difference.ToString() + "  -  MAX: "
      + maxReading.ToString());
    Thread.Sleep(200);
}
```

Using your breathalyzer

With everything coded and set up, running the code should have the screen displaying values. When blowing on it with normal (non-alcohol) breath the values will fluctuate a bit, and may even go slightly below 100 percent, but that is fine. Blowing on it after having a drink, however, will have the value skyrocketing. In order to get a better accuracy while comparing your intoxication level with your friends, be sure to keep the following in mind:

- Sipping a drink just before blowing will skew the reading. You should wait about 10 minutes after your last drink before taking a reading.
- Smoking can also change the result unpredictably.
- After taking a reading, the sensor will take a few minutes to fall back down to the baseline. So just press the Netduino's onboard button to reset it once after blowing and by the time the "warming up" process has completed the value will be back to normal. Basically, reset the Netduino after each reading.

According to the datasheet for this sensor, it is recommended that you let it run for 24 to 48 hours when you first get it. This is to "burn" it in, and only needs to be done once.

Other ideas and hints

Following are a few ideas and hints that you can consider while creating your project:

- A common way to power the Netduino is with a 9V battery. Doing this will allow you to take your breathalyzer anywhere! Keep in mind that the sensor uses a lot of power, so bring an extra 9V battery and disconnect battery power from the Netduino when not in use.
- Once powered with a battery, create a custom case around it to make it easily transportable.
- Find someone that is completely drunk and get a reading from them. Then use that as a maximum value and work out a rough drunk percent in code.
- Connect up some colored LEDs and turn them on/off based on how high the reading is. For example, red for drunk, and green for sober.
- Many similar sensors exist that can be used to detect everything from flammable gas to smoke, and are used in exactly the same way as this sensor.

Summary

In this chapter, you learned how to power a sensor with 5V, and then divide the analog return voltage to get it down to 3.3V for the Netduino. Everything else was built on top of the knowledge that you have gained from previous chapters.

In the next chapter, we will learn how to lock your doors with code!

8

Hide Yo' Kids, Hide Yo' Wife – with Automated Locks!

One of the most important parts of automating a house is security. There are countless things we could make to keep people out, but an electronic door lock should probably be first. Also, most importantly, using a keypad to unlock your door is pretty awesome.

In this chapter we will cover:

- Understanding, connecting, and using a keypad
- Using a servo creatively

Things you need

- Netduino
- Breadboard
- Membrane/matrix keypad (we use a 3 x 4 version here, but yours can vary)
- 4 X 10K ohm resistors
- Strong servo (a TowerPro MG995 will do)
- Standard bolt lock

The project setup

Along with setting up the electronics, this project has some mechanics that we need to go over first.

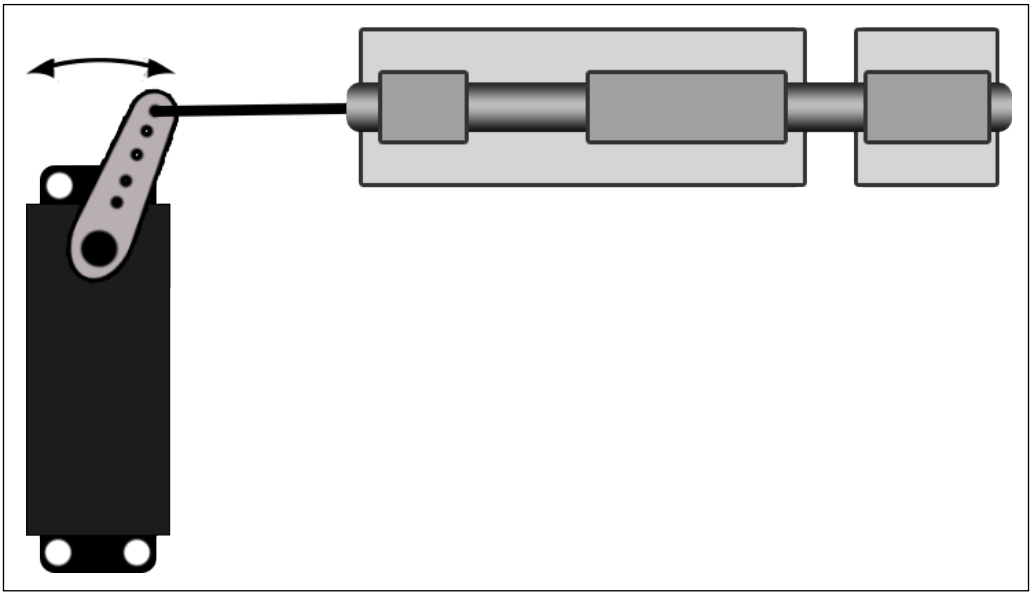
The lock

Some people will automatically go off to eBay and look for an electric lock to purchase for a project like this, and that is a perfectly valid solution. However, most of the fun with making little devices is that you didn't just buy them off the shelf. It goes without saying that pre-made solutions, although considerably more expensive, will be more reliable.

The setup of the lock is a little tricky to explain because every situation is going to vary, and no single solution is best for everyone. What it basically means is that this chapter will only spend a small amount of time talking about the mechanics of the lock, and more time in explaining the code and electronics.

At any hardware store you can pick up a cheap sliding bolt lock. These vary in shape and size, but the basic principle of a long bolt sliding inside a tube will apply to all of them. Depending on the one you get, there could be a better way to rig it up. A lot of them have a removable handle, screw it out or grind it off, which will let the bolt move freely. Once the handle is off, get a strong piece of wire that will not bend easily, and connect it to the back of the bolt. A good way to connect it is to drill a small hole through the bolt and thread the wire through, which will allow the wire to angle slightly up/down when the servo moves.

Once the wire is attached to the bolt, simply attach it to an arm on the servo. When the arm moves backwards and forwards the bolt will slide in and out:

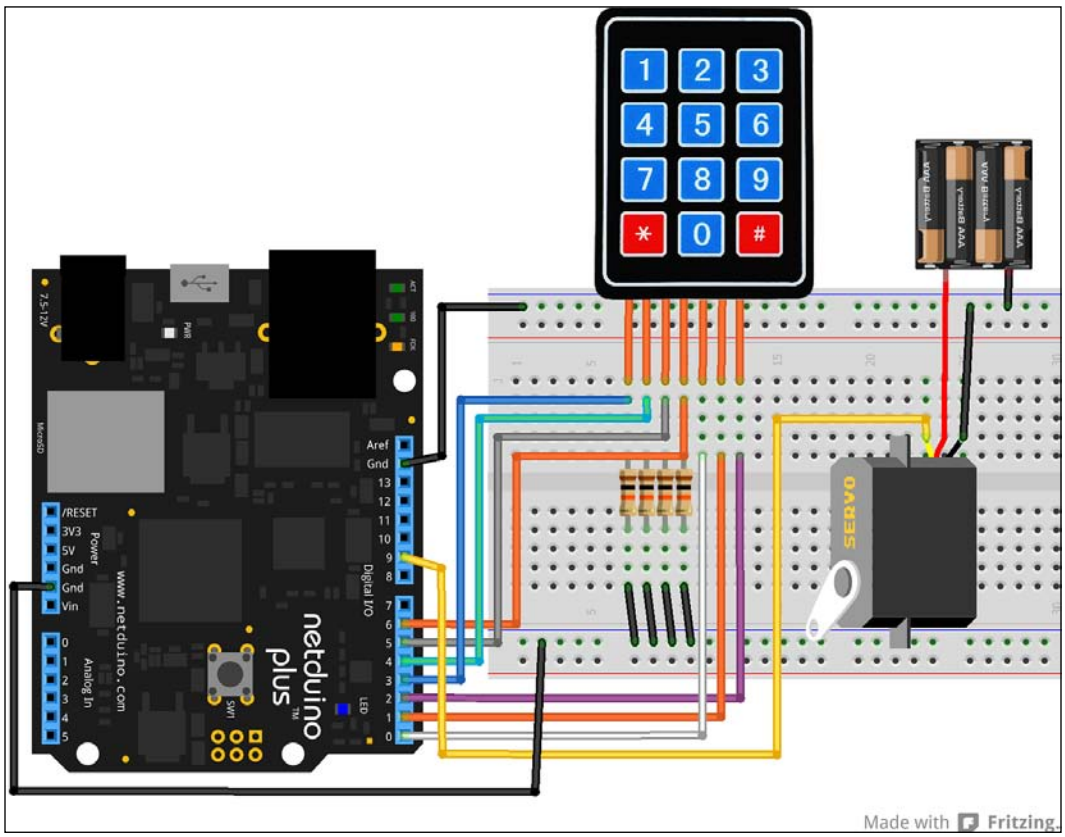


The hardest part is going to be attaching the servo to the door. This can be achieved by sticking/screwing the front face of the servo (below the moving arm) to a small block of wood, and attaching that to the door.

The preceding method is definitely not the best way; however, it is definitely the easiest and it is universal. Go ahead and create one that works for you!

Electronics

Although the electronics may look complicated at first, nothing is new, there are just more wires now! The servo is connected to the Netduino to control it, but the power is from a separate battery pack (just like powering the LEDs). Also, remember to connect the ground wire from the battery pack with the ground from the Netduino:



The keypad is a little harder to understand. It is split up into rows and columns, so it has seven pins (on its ribbon cable) because there are (four rows) + (three columns). The first four pins are the rows, and the last three are the columns. The four row pins have pull-down resistors on them.

To determine if a button is pressed, you need to know what row and column you are looking at, and then see if current is flowing between those pins. For example, if we want to check if *4* is being pressed, we need to see whether pin 5 (column 1) and pin 2 (row 2) are connected. To see if they are connected with the Netduino, we put 3.3V on pin 5, and then read the voltage on pin 2 – if pin 2 has a higher reading then we know that they are connected.

This means that we need to do the following process every time we want to check which buttons are pressed:

1. Apply power to pin 5 (column 1).
2. Individually check the reading on pins 1 to 4 to see if any of the buttons (*1*, *4*, *7*, or *) in column 1 are pressed.
3. Apply power to pin 6 (column 2).
4. Redo step 2 to check if any buttons (*2*, *5*, *8*, or *0*) are pressed.
5. Apply power to pin 7 (column 3).
6. Redo step 2 to check if any buttons (*3*, *6*, *9*, or #) are pressed.

If your keypad has a different number of rows or columns, just adjust the code in the next section accordingly. To work out which pins are the rows and which are the columns, use a multimeter set to the "continuity" setting.

The Netduino code

Remember using the .NET Micro Framework Toolbox subframework? Well, that library actually has some code to help with using a keypad like this. However, it is important too that we go over some more complicated code occasionally, even if it feels a bit like reinventing the wheel. An advantage of doing it yourself is that you will know what is going on, as opposed to attempting to read someone else's code which may appear like a long-lost Klingon dialect:

1. Create a new **Netduino Plus 2 Application**.
2. The whole lock mechanism is pretty useless if we can't move the servo, so add the servo library as in *Chapter 6*, *You've Got Mail, and Here's a Flag to Prove It*.

3. In `Program.cs` we need to declare a few class variables. Just above the `Main` method add these lines:

```
private const string _password = "1337";
private static string _currentlyEntered = "";
private static Servo _servo;
private static bool _isLocked = false;
```

   The `password` variable contains the code that will unlock the door. Change that to whatever you want.

4. Now, within the `Main` method we can instantiate the servo and create some more variables. As mentioned earlier, we will be powering up each column pin individually and then reading each row pin. So we will declare the columns as `OutputPort` and the rows as `InputPort`:

```
InterruptPort btn = new InterruptPort(Pins.ONBOARD_BTN,
  false, Port.ResistorMode.Disabled, Port.InterruptMode.
    InterruptEdgeHigh);
btn.OnInterrupt += new NativeEventHandler(btn_OnInterrupt);

_servo = new Servo(PWMChannels.PWM_PIN_D9);
OutputPort[] columns = new OutputPort[3];
columns[0] = new OutputPort(Pins.GPIO_PIN_D0, false);
columns[1] = new OutputPort(Pins.GPIO_PIN_D1, false);
columns[2] = new OutputPort(Pins.GPIO_PIN_D2, true);

InputPort[] rows = new InputPort[4];
rows[0] = new InputPort(Pins.GPIO_PIN_D3, true,
  Port.ResistorMode.Disabled);
rows[1] = new InputPort(Pins.GPIO_PIN_D4, true,
  Port.ResistorMode.Disabled);
rows[2] = new InputPort(Pins.GPIO_PIN_D5, true,
  Port.ResistorMode.Disabled);
rows[3] = new InputPort(Pins.GPIO_PIN_D6, true,
  Port.ResistorMode.Disabled);
```

5. An `InterruptPort` method is a way to create an event handler for a button press. It is the same as manually checking, but requires a lot less code. It simply reads every loop and will fire the `OnInterrupt` handler when it is pressed. In this case we use it to override the default action of the on-board button (which usually restarts the Netduino) to lock the door. An `InputPort` variable allows us to read whether current is being sent to a pin or not (you may see this referred to as `HIGH` and `LOW`). Basically, it is the opposite of what an `OuputPort` variable does. We store all these instances in arrays so we can loop through them easily.

6. Given its row and column, we need a way of easily knowing what the character on a pressed button is. For this we can create a **jagged array** (2D arrays are not supported in **.Net Micro Framework** (**.NETMF**)):

```
char[][] characters = new char[4][];
characters[0] = new char[] { '1', '2', '3' };
characters[1] = new char[] { '4', '5', '6' };
characters[2] = new char[] { '7', '8', '9' };
characters[3] = new char[] { '*', '0', '#' };
```

7. This next code is the main loop which does all the actual work. Each block of code goes after the previous one, and is split up to explain it better. Sometimes when lifting the button after pressing, some erroneous presses will come through. To avoid that, we set a number of loops that must happen after a press before another one can happen. This code declares the variables to do that, and has a loop to loop through each column:

```
int loopsSinceLifted = Int32.MaxValue;
while (true) {
  bool pressed = false;
  for (int c = 0; c < columns.Length; c++){
```

Inside the column loop, we loop through the columns again to turn `OFF` the two pins we are not using and turn on the single one that we are:

```
for (int cInner = 0; cInner < columns.Length; cInner++){
  columns[cInner].Write(cInner == c);
}
```

Now that the correct column pin is turned `ON`, we loop each row and see if any of them are receiving current (the `.Read()` method returns `true` if it is). If so, check that the code has looped at least 10 times since the last press, and then call a method to handle the press:

```
for (int r = 0; r < rows.Length; r++){
  if (rows[r].Read()) {
    pressed = true;
    if (loopsSinceLifted > 10) {
      HandlePress(characters[r][c]);
      loopsSinceLifted = 0;
      break;
    }
  }
}
```

If there was a press then we can stop looping, and if there was no press then increment the count by 1. Each loop has a 20 ms sleep time:

```
  if (pressed) {
    break;
  }
}
  if (!pressed) {
    loopsSinceLifted++;
  }
  Thread.Sleep(20);
  }
}
```

8. Remember the code we used to override the default action of the on-board button? We haven't actually defined what we want it to do instead, so below the `Main` method put this event handler:

```
private static void btn_OnInterrupt(uint data1, uint data2,
  DateTime time) {
    HandlePress('*');
}
```

When we press the on-board button it will be the same as pressing the * key on the keypad, which will lock the door.

9. Just a few lines left! Next we add the method that will handle the key presses. If the user presses the * key then the currently entered code will be cleared and the door will be locked. And if # is pressed then the code will be compared with the correct code:

```
private static void HandlePress(char character) {
  if (character == '*') {
    if (!_isLocked) {
      Lock();
    }
    _currentlyEntered = "";
  }
  else if (character == '#') {
    if (_currentlyEntered == _password) {
      Unlock();
    }
    else {
      _currentlyEntered = "";
    }
  }
  else {
    _currentlyEntered += character;
  }
}
```

10. The last two methods simply move the servo to open and close the lock. The number of degrees turned may differ based on your lock and the type of servo you use (some will turn 180 degrees, while others may only turn 90 degrees) :

```
private static void Lock() {
  _servo.Degree = 90;
}

private static void Unlock() {
  _servo.Degree = 25;
}
```

Usage

After attaching the lock to the door, and the running wires to the other side of the door to mount the keypad, you can start using it!

To unlock the door, type your code and then press #. If you make a mistake, just press * to start over. If you want to lock the door, press the * key.

Once you are inside and need to lock the door, just press the on-board button on the Netduino.

Other ideas and hints

- Currently, the only way to lock the door from inside is using the on-board button, which can be inconvenient if you want to put the Netduino in a neat enclosure. Wire up a dedicated button instead.
- To get even more geek-cred, add a fingerprint reader instead of the keypad.
- Add a Bluetooth module and use your smartphone to lock and unlock the door.

Summary

This chapter was quite heavy on code, but then you must have handled it just fine! You learned how to work with switches and inputs. And even how to use a relatively complex matrix keypad. Finally, you got your hands dirty and probably caused mild structural damage to your door – but all in the name of SCIENCE!

In the next chapter we will be taking security a step further by creating a motion alarm.

9
Rogue Alert: Detecting Intruders in Your House or Fridge

So what's next in our quest? Detecting intruders! In this chapter, we will use a **Passive Infrared** (**PIR**) **sensor**—the ones used in home security systems—to detect movement, and then e-mail a set address with a notification. This could be used for a few different reasons, such as protecting your stash of dorm-room snacks from "friends", or getting notified when someone is in your house. But let's be real, the single greatest threat to humanity currently is zombies! Knowing when they enter your property can buy valuable crossbow-loading time, and may well save your life.

In this chapter we will cover:

- Connecting and using a PIR sensor to detect movement
- Sending an e-mail with SMTP
- Using a flex sensor to detect a door opening
- Making a trip-wire
- Using a piezo buzzer

Things you need

Following is the hardware required for this project:

- Netduino (must be a Plus or Plus 2)
- Internet and Ethernet cable to connect the Netduino
- Breadboard

- IR Infrared Motion Detection sensor
- Flex sensor
- 10K ohm resistor
- Piezo buzzer

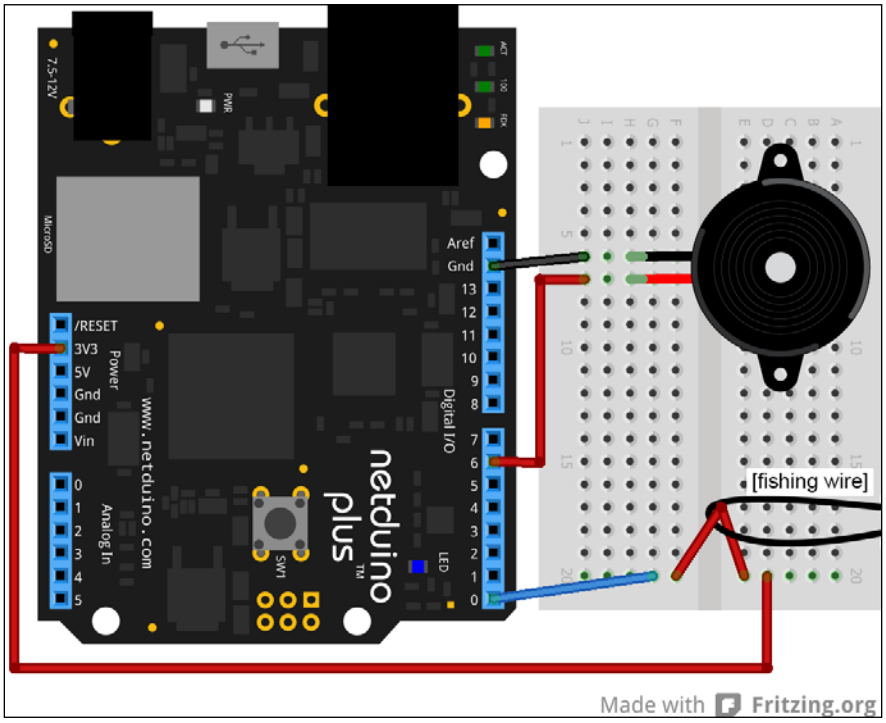
Although many variants of the PIR sensor exist, the one from DealExtreme is great because the output is 3.3V, and you don't need to add a resistor to use it.

Setting up the triggers

We will go over three different "triggers" for our system, first by showing the electronics setup, then by going over the code for each.

The trip wire setup

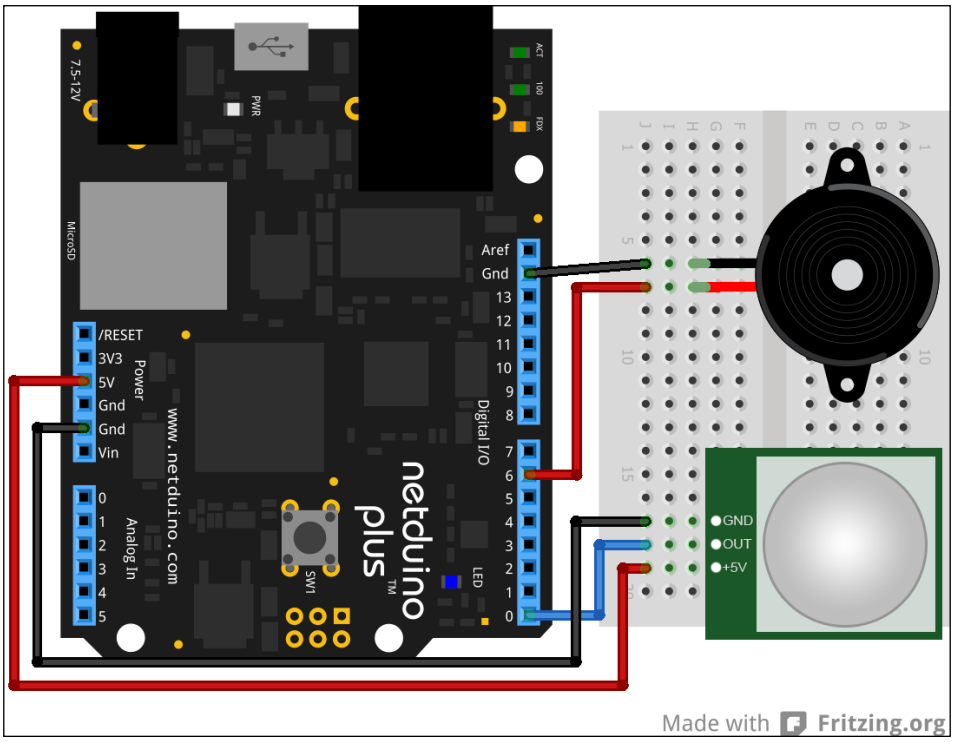
One of the simplest methods of intruder detection is the age old trick of putting a fishing wire across an entrance and attaching it to a bell.



This is a slightly modernized version of that, but is still really simple. Essentially, we just have a wire going from the **3V3** pin to a digital pin. We constantly read the digital pins output to see if it is still receiving current, and when it stops we know that the wire has been tripped over and pulled out. It probably isn't a good idea to have wires yanked directly out of the Netduino though, so we run them to a breadboard where we can attach a small secondary wire to the fishing line across the entrance.

The PIR sensor setup

A Passive Infrared sensor (also called a Pyroelectric Infrared sensor) works by detecting spikes in the infrared radiation levels of an area. To simplify this, just think of it as being a fancy sensor that can *see* heat given off by people, animals, and so on.

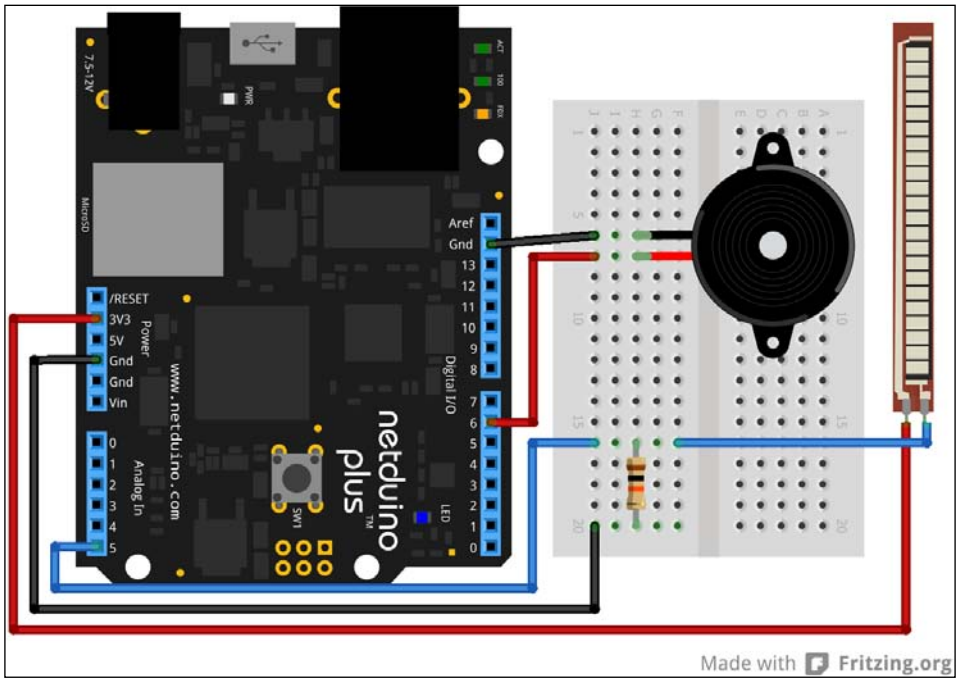


We power it from the **5V** and **GND** pins. When the sensor detects movement it will output a HIGH value (3.3V) on the **OUT** pin, and when there is no movement then it will have a **LOW** value. Be careful to read the datasheet of the sensor you buy (if it differs from the one used here) to make sure the output is 3.3V.

The sensor has two twistable knobs. The left one controls the amount of time that the sensor will stay *high* after a movement. Turn it totally counterclockwise because our code will control that. The right knob controls the sensitivity, which you should work out through trial and error. Counterclockwise will turn the sensitivity down, and clockwise will turn it up.

The flex sensor setup

Occasionally, it might not be practical to use a PIR sensor or trip wire, and this is where a flex sensor comes in. A flex sensor does exactly what you expect from the name—it measures how much it is being bent. We can stick this on the hinge side of a door so that we can determine when the door is opened. The side with the rectangles should be facing the door so that the *FLEX SENSOR BY SPECTRA SYMBOL* text is visible.



The sensor is connected to **3V3** and an AnalogIn port, with a 10k ohm resistor also tying it to GND.

Code for the triggers

The code will be split into three different parts (for the different sensors), and you can choose which code to use based on which sensor you are using. Or just combine them to use all three methods at the same time. But first, let's get the project set up and put in the code that is common for all of them:

1. Create a new Netduino Plus 2 Application.
2. We will be using a PWM and an Analog input. So just as you have done before, reference `SecretLabs.NETMF.Hardware.PWM` and `SecretLabs.NETMF.Hardware.Analoginput`.
3. To send an e-mail, we will be using the SMTP library from the .NET Micro Framework Toolkit. Reference the 4.3 versions of these DLLs: `Toolbox.NETMF.NET.Core`, `Toolbox.NETMF.NET.SMTP_Client`, and `Toolbox.NETMF.NET.Integrated`.
4. Add the `using` statements for all of these to the top of `Program.cs`:

```
using NPWM = SecretLabs.NETMF.Hardware.PWM;
using NAnalog = SecretLabs.NETMF.Hardware.AnalogInput;
using Toolbox.NETMF.NET;
```

5. Now let's declare variables for the on-board LED and the pin that the buzzer is connected to. Also, we make the LED flash for 20 seconds to allow us time to leave the room:

```
        private static NPWM _buzzer;
        private static OutputPort _led;
        public static void Main()
        {
            _buzzer = new NPWM(Pins.GPIO_PIN_D6);
            _led = new OutputPort(Pins.ONBOARD_LED, false);

            bool ledOn = true;
            for (int i = 0; i < 100; i++)
            {
                _led.Write(ledOn);
                ledOn = !ledOn;
                Thread.Sleep(200);
            }
  //code sections here
        }
```

Notice the comment where the code in the next sections should appear.

6. The last parts of this common code are the methods that will actually be used to sound the buzzer and send out the alert e-mail. We use a single method to call both of those actions:

```
        private static void TriggerAlarm()
        {
            SendEmail();
            SoundBuzzer(100);
            _led.Write(true);
            Thread.Sleep(10000);
            SoundBuzzer(0);
            _led.Write(false);
        }

        private static void SendEmail()
        {
            SMTP_Client.MailMessage message = new SMTP_Client.
MailMessage("ROGUE ALERT!", "An intruder has been detected");
            SMTP_Client client = new SMTP_Client(new
IntegratedSocket("smtp.yourISP.net", 25));
            client.Send(message, new SMTP_Client.
MailContact("your.sexy@netduino.com"), new SMTP_Client.
MailContact("your.email@address.com"));
        }

        private static void SoundBuzzer(uint val)
        {
            _buzzer.SetDutyCycle(val);
        }
```

7. Sending an e-mail is very simple using the .NET Micro Framework Toolbox. The hardest part will be finding out what your ISP's SMTP settings are.

   Although your e-mail provider may provide this, your actual Internet provider will almost always have an open server to send mail through. Besides the SMTP URL, the only other thing you need to change in the code is `your.email@address.com`, which should be the address to which you want the code to send an e-mail. You can change the sender address too, which can really just be anything.

   Also notice that we sound the buzzer for 10 seconds. This also acts as a buffer time before the sensors will get triggered again.

Next, we have the specific code for each method of detection. The code given in the following sections should be inserted where the comment is in the `Main` method.

The trip wire code

To detect when the fishing line is tripped over (pulling out the wire) all we need to do is detect when there is no longer a *high* current on the pin. We do that by simply waiting for the `Read` method to return `false`:

```
            InputPort tripwire = new InputPort(Pins.GPIO_PIN_D0, true,
Port.ResistorMode.Disabled);
            while (true)
            {
                if (!tripwire.Read())
                {
                    TriggerAlarm();
                }
            }
```

The PIR sensor code

The following code is the exact opposite of the previous code. This PIR sensor will make the pin *high* when there is movement:

```
            InputPort pir = new InputPort(Pins.GPIO_PIN_D0, true,
Port.ResistorMode.Disabled);
            while (true)
            {
                if (pir.Read())
                {
                    TriggerAlarm();
                }
            }
```

The flex sensor code

This one is slightly different to the previous two because it is an Analog Input. We get an integer value when we read the pin, and from that can determine how bent/flexed the sensor is. The specific value that triggers the alarm may differ based on your needs (if you have a multimeter, you can check the output of the sensor as you flex it and use that value as a starting point. In my setup, 270 works.):

```
            NAnalog flex = new NAnalog(Pins.GPIO_PIN_A5);
            while (true)
            {
                if (flex.Read() < 270)
                {
                    TriggerAlarm();
                }
            }
```

Go wild

Currently this system is pretty limited because you can't actually tell who/what is triggering the alarm, and besides hearing a rather annoying sound, nothing happens to them/it.

I don't want to put any ideas in your head such as adding weaponry to your setup—so I won't—but you should get creative with it and explore ways to expand the functionality.

Remember, time is running out and they are coming!

Summary

In this chapter, you learned how easy it is to send an e-mail using an existing library, and used a few different sensors to detect intruders in various ways.

As you have seen, there are a few main principles to understand, but once you know those, using a wide range of components suddenly becomes very easy.

In the final chapter, we will save lives with the *elixir of life*.

10
Saving Lives – Automated Watering

If video games have taught us anything, it is that the best defense against zombies is plants. The problem is that plants will die without water, and manually watering them is far more work than any of us deserve. So let's get a Netduino to do it!

In this chapter we will cover:

- Connecting and using a **soil moisture sensor** to detect how dry the soil is
- Using a servo to turn the water valve to allow water to flow

Things you need

- Netduino
- Soil moisture sensor
- Strong servo (a TowerPro MG995 will do)
- 4 x AA Battery pack
- Water valve that is easy to turn and only requires about 90 degrees to go from fully open to fully closed
- Some pipes that fit the valve

The project setup

As with the automated lock, this project has a mechanical aspect along with the code and the electronics.

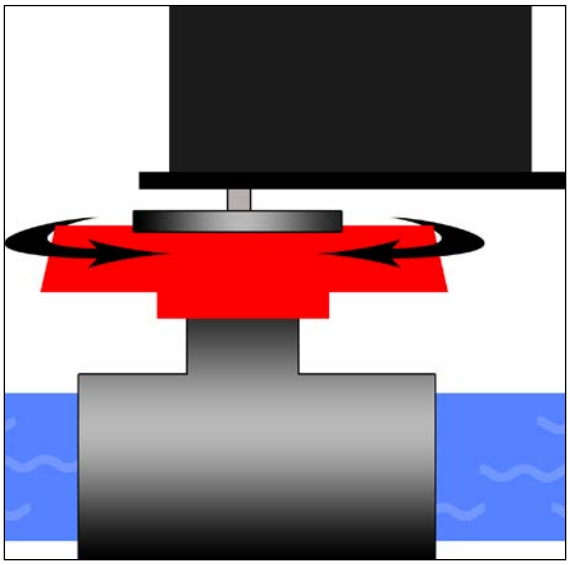
The valve

The most complicated part of this project is getting the water to actually flow and stop on command. There are a few ways to do this, and you should work out which one is best for your situation and wallet.

A quick and easy way would be to buy an electric water valve which will open or close when you power it `ON`. These usually run on higher voltages (12V), so will need to be connected to a separate power supply. Also keep in mind that most of these require a high-pressure system which means that they will only work when connected to a tap and not just with a bucket of water.

The most universal way, however, is to use a strong servo (like the one we used in our automated lock in *Chapter 8, Hide Yo' Kids, Hide Yo' Wife – with Automated Locks!*) to turn a valve. Most hardware stores will have a small valve that is easy to turn and only has 90 degrees of movement. These are often used as shut-off valves near a washing machine. We don't want the type that can turn multiple times (like a tap).

With valve and servo in-hand, we can connect them up:



An easy way to strap the valve and servo together is to cable-tie them both, flat to a board.

The nice thing about a solution like this is that it will not only work when connected to a tap, but will also work when connected to a bucket/gravity system. If you have enough space, you can create a rain collector which feeds water into the bucket, making it eco-friendly too!

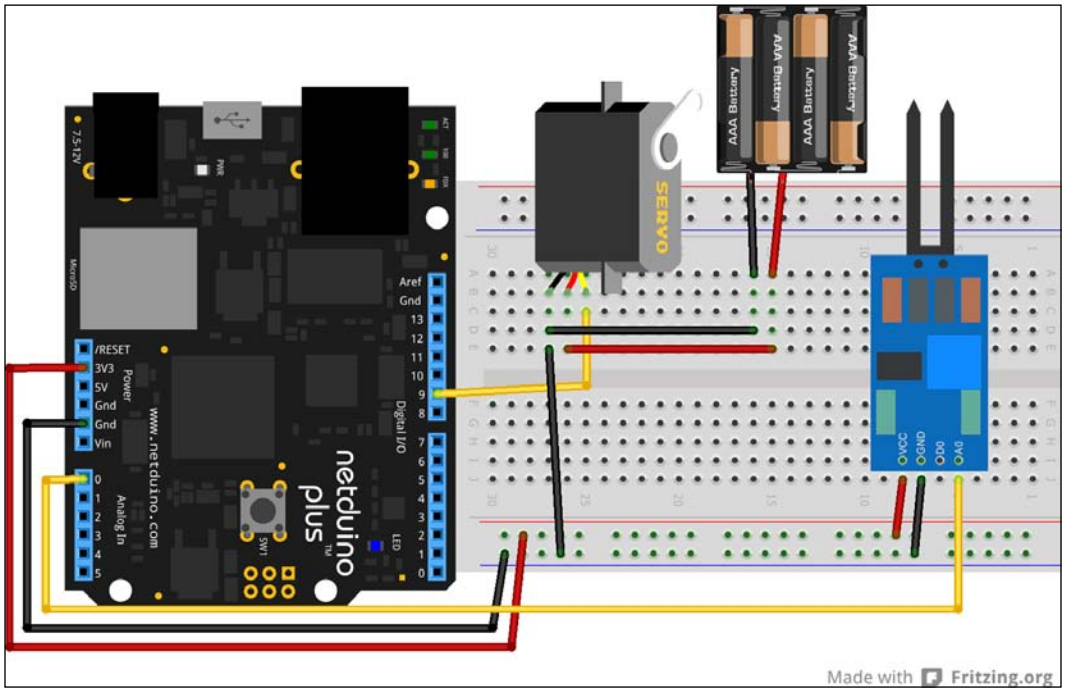
Some countries have laws against collecting rain water (I know, right?), so be sure to check with local authorities before doing this.

Keep the servo's arms disconnected from the valve until your code is all working. Once the code is done, let it set the servo to the closed position, then only tie the servo arms to the valve. This will make sure that the servo isn't damaged by being in the wrong position the first time it is connected.

Electronics

The electronics setup is really easy, and has nothing we haven't done before.

The servo is powered by a separate battery pack, and the ground is tied with the Netduino ground. Then the soil moisture sensor is powered by 3.3V from the Netduino, and we connect up the A0 pin to an **Analog Input** pin on the Netduino:



The soil sensor has a digital (D0) and analog (A0) pin. We use the analog pin which will tell us the exact value. The digital pin, on the other hand, will either return `true` or `false`, based on whether the moisture level is over/under what is defined by the turnable knob on the sensor. Notice the two spikes on the soil sensor? Those get poked into the ground.

The Netduino code

With everything plugged in and connected together, all that remains is some code:

1. Create a new **Netduino Plus 2 Application**.
2. Reference the **SecretLabs.NETMF.Hardware.AnalogInput** DLL as before.
3. Add the servo API class that we edited in *Chapter 2*, *You've Got Mail, and Here's a Flag to Prove It*.
4. Now that we've included everything we need, add the following at the top of `Program.cs`:

   ```
   using AINP = SecretLabs.NETMF.Hardware.AnalogInput;
   using Servo_API;
   ```

5. Above the `Main` method, we declare a few variables:

   ```
   private static Servo _valveServo;
   private const double _valveOpenPosition = 130;
   private const double _valveClosedPosition = 180;
   ```

   The two servo values are used to set the closed and open position of the servo in degrees. Depending on your valve, you may need to change these values after some trial and error. *Remember*! Never test with the servo arms connected to the valve; or else the servo may get damaged.

6. The `Main` method is pretty straight forward, and you can tweak the behavior. We first initiate the servo, and create a reference to the analog pin that the moisture sensor is connected to. Then we open and close the valve just to make sure that it is able to move and that the system starts off with it closed:

   ```
   public static void Main() {
     _valveServo = new Servo(PWMChannels.PWM_PIN_D9);
     AINP moistureSensor = new AINP(Pins.GPIO_PIN_A0);
     OpenValve();
     CloseValve();
     while (true) {
       int moisture = moistureSensor.Read();
       Debug.Print(moisture.ToString());
   ```

```
    if (moisture > 700) {
      ActivateWater();
    }
    Thread.Sleep(1000);
  }
}
```

Within the loop, we continually check the value of the moisture sensor. The sensor will return close to **0** for a perfect connection (fully water-logged plant), and close to **1024** for no connection (totally dry soil). When the value goes above **700** it means that the plant is getting too dry, so we need to water it. The loop has a 1 second sleep on it, but this could be set much higher to save power.

7. When the soil was too dry we called a method to water the plant. This method will open the valve (to let water flow), leave it open for 5 seconds, then close it. After closing, we wait another 60 seconds to let the water seep into the soil and spread around before resuming the loop:

```
private static void ActivateWater() {
  OpenValve();
  Thread.Sleep(5000);
  CloseValve();
  Thread.Sleep(60000);
}
```

8. Finally, we have the methods which do the actual opening and closing of the valve:

```
private static void OpenValve() {
  _valveServo.Degree = _valveOpenPosition;
  Thread.Sleep(200);
  _valveServo.disengage();
}

private static void CloseValve() {
  _valveServo.Degree = _valveClosedPosition;
  Thread.Sleep(200);
  _valveServo.disengage();
}
```

In each of those methods we first set the value, wait for 200 ms to allow it to get to the right angle, and then disengage the servo. Because the servo will struggle a bit on the last few degrees of opening and closing the valve, it might vibrate and potentially get damaged, so disengaging the servo means that it will not attempt to keep the angle once it is there.

Usage

Once everything is done, you can test out the system. To try it out, get some water and place the prongs of the soil sensor in it (note that only the prongs must go into the water/soil, not the whole unit). Then plug the Netduino in and wait for it to boot up. Removing the prongs from the water will make the servo turn the valve to open it, and then close after 5 seconds.

If that works then you can go ahead and test it in real soil, with a real plant. The prongs should be placed quite close to the roots of the plant, and if the plant is outside, make sure to properly waterproof the Netduino.

Depending on your environment and plants you may want to adjust the value of 700 (that was used as the dry threshold in the code) to something more suitable.

Other ideas and hints

- This exact setup can be used to keep your pet's water bowl filled up. Be sure to tape everything up to avoid your dog inevitably gnawing on the wires.
- With some trial and error, this sensor could be used as a rough rain meter by placing the prongs at the bottom of a funnel in the rain. The more water, the lower the value will be.
- Use your e-mailing skills to make the Netduino e-mail you a status report once a week with the data about the plant, and possibly even log the moisture levels.

Summary

In this chapter you learned how to control water with your Netduino, and how to measure the moisture in soil (or a water bowl) with a soil sensor. In doing so, you have single-handedly saved the lives of countless plants and animals in your future! No longer can your parents look down on you for not being a surgeon and no longer will you be referred to as an "irresponsible juvenile" by your in-laws!

Well, that's it. You've successfully come this far! Find a mirror, twist around, and pat yourself on the back, because you are pretty epic!

Index

L

N

P

R

S

T

V

W

Thank you for buying
Netduino Home Automation Projects

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

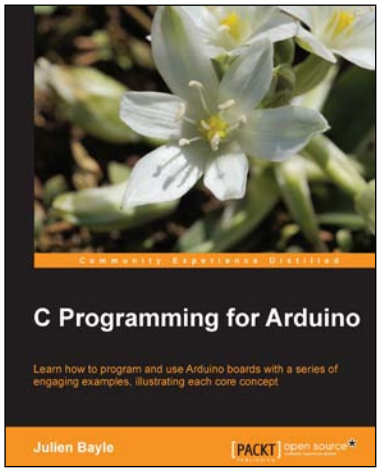
Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike.
For more information, please visit our website: `www.packtpub.com`.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

[PACKT]
PUBLISHING

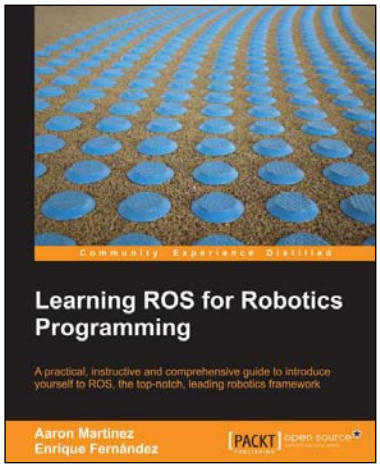


C Programming for Arduino

ISBN: 978-1-84951-758-4 Paperback: 512 pages

Learn how to program and use Arduino boards with a series of engaging examples, illustrating each core concept

1. Use Arduino boards in your own electronic hardware and software projects
2. Sense the world by using several sensory components with your Arduino boards
3. Create tangible and reactive interfaces with your computer
4. Discover a world of creative wiring and coding fun



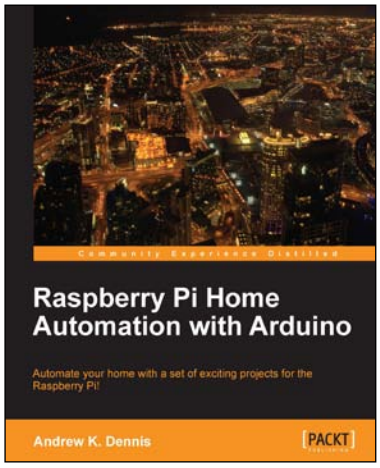
Learning ROS for Robotics Programming

ISBN: 978-1-78216-144-8 Paperback: 374 pages

A practical, instructive and comprehensive guide to introduce yourself to ROS, the top-notch, leading robotics framework

1. Model your robot on a virtual world and learn how to simulate it
2. Carry out state-of-the-art Computer Vision tasks
3. Easy to follow, practical tutorials to program your own robots

[PACKT]
PUBLISHING



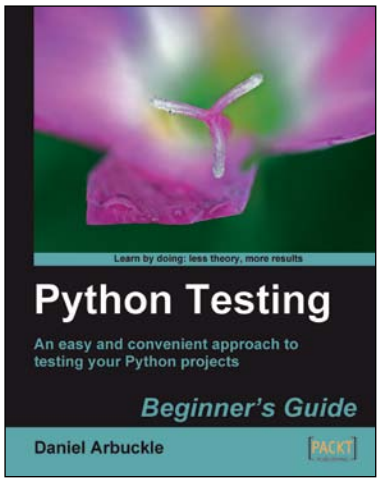
Raspberry Pi Home Automation with Arduino

ISBN: 978-1-84969-586-2 Paperback: 176 pages

Automate your home with a set of exciting projects for the Raspberry Pi!

1. Learn how to dynamically adjust your living environment with detailed step-by-step examples
2. Discover how you can utilize the combined power of the Raspberry Pi and Arduino for your own projects
3. Revolutionize the way you interact with your home on a daily basis

---



Python Testing: Beginner's Guide

ISBN: 978-1-84719-884-6 Paperback: 256 pages

An easy and convenient approach to testing your Python projects

1. Covers everything you need to test your code in Python
2. Easiest and enjoyable approach to learn Python testing
3. Write, execute, and understand the result of tests in the unit test framework
4. Packed with step-by-step examples and clear explanations