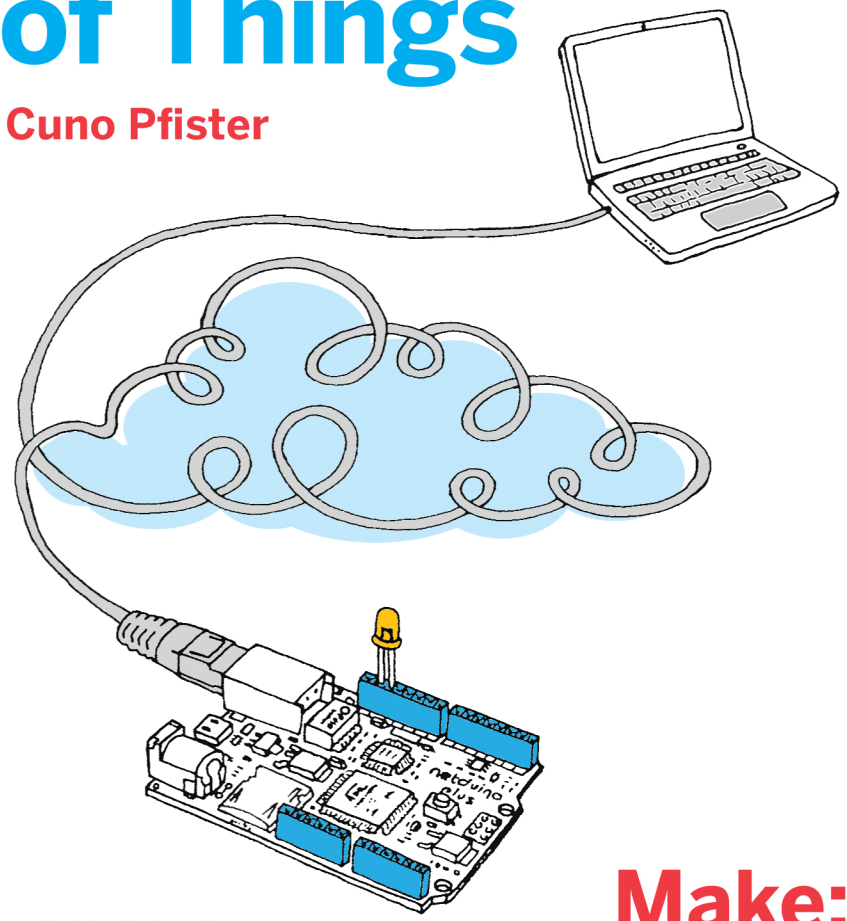
Make: PROJECTS

CONNECTING SENSORS AND MICRO-CONTROLLERS TO THE CLOUD

Getting Started with the Internet of Things

Cuno Pfister



O'REILLY

Make: makezine.com

O'Reilly Ebooks—Your bookshelf on your devices!

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

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

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

Getting Started with the Internet of Things

by Cuno Pfister

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

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

Print History: May 2011: First Edition.

Editor: Brian Jepson
Production Editor: Jasmine Perez
Copyeditor: Marlowe Shaeffer
Proofreader: Emily Quill
Compositor: Nancy Wolfe Kotary
Indexer: Angela Howard
Illustrations: Marc de Vinck
Cover Designer: Marc de Vinck

ISBN: 978-1-4493-9357-1

[LSI]

Preface

One of the most fascinating trends today is the emergence of low-cost *microcontrollers* that are sufficiently powerful to connect to the Internet. They are the key to the *Internet of Things*, where all kinds of devices become the Internet's interface to the physical world.

Traditionally, programming such tiny *embedded* devices required completely different platforms and tools than those most programmers were used to. Fortunately, some microcontrollers are now capable of supporting modern software platforms like .NET, or at least useful subsets of .NET. This allows you to use the same programming language (C#) and the same development environment (Visual Studio) when creating programs for small embedded devices, smartphones, PCs, enterprise servers, and even cloud services.

So what should you know in order to get started? This book gives one possible answer to this question. It is a *Getting Started* book, so it is neither an extensive collection of recipes (or design patterns for that matter), nor a reference manual, nor a textbook that compares different approaches, use cases, etc. Instead, its approach is "less is more," helping you to start writing Internet of Things applications with minimal hassle.

The Platforms

The *.NET Micro Framework* (NETMF) provides Internet connectivity, is simple and open source (Apache license), has hardware available from several vendors, and benefits from the huge .NET ecosystem and available know-how. Also, you can choose between Visual Studio (including the free Express Edition) on Windows, and the open source Mono toolchain on Linux and Mac OS X.

There is an active community for NETMF at *http://www.netmf.com/Home.aspx*. The project itself is hosted at *http://netmf.codeplex.com/*.

Netduino Plus (*http://www.netduino.com/netduinoplus*) is an inexpensive NETMF board from *Secret Labs* (*http://www.secretlabs.com*). This board makes Ethernet networking available with a price tag of less than $60. It has the following characteristics:

- A 48 MHz Atmel SAM7 microcontroller with 128 KB RAM and 512 KB Flash memory
- USB, Ethernet, and 20 digital I/O pins (six of which can be configured optionally for analog input)
- Micro SD card support
- Onboard LED and pushbutton
- Form factor of the Arduino (*http://www.arduino.cc/*); many Arduino *shields* (add-on boards) can be used
- .NET Micro Framework preprogrammed into Flash memory
- All software and hardware is open source

There is an active community for the Netduino Plus (and NETMF) at *http://forums.netduino.com/*. All the examples in this book use the Netduino Plus.

How This Book Is Organized

The book consists of three parts:

- Part I, Introduction

  The first part tells you how to set up the development environment and write and run a "Hello World" program. It shows how to write to output ports (for triggering so-called *actuators* such as LED lights or motors) and how to read from input ports (for *sensors*). It then introduces the most essential concepts of the Internet of Things: HTTP and the division of labor between clients and servers. In the Internet of Things, devices are programmed as clients if you want them to push sensor data to some service; they are programmed as servers if you want to enable remote control of the device over the Web.

» Part II, Device as HTTP Client

  The second part focuses on examples that send HTTP requests to some services—e.g., to push new sensor measurements to the Pachube service (*http://www.pachube.com*) for storage and presentation.

» Part III, Device as HTTP Server

  The third part focuses on examples that handle incoming HTTP requests. Such a request may return a fresh measurement from a sensor, or may trigger an actuator. A suitable server-side library is provided in order to make it easier than ever to program a small device as a server.

» Appendix A, Test Server

  This contains a simple test server that comes in handy for testing and debugging client programs.

» Appendix B, .NET Classes Used in the Examples

  This shows the .NET classes that are needed to implement all examples, and the namespaces and assemblies that contain them.

» Appendix C, Gsiot.Server Library

  This summarizes the interface of the helper library `Gsiot.Server` that we use in Part III.

Who This Book Is For

This book is intended for anyone with at least basic programming skills in an object-oriented language, as well as an interest in sensors, microcontrollers, and web technologies. The book's target audience consists of the following groups:

» Artists and designers

  You need a prototyping platform that supports Internet connectivity, either to create applications made up of multiple communicating devices, or to integrate the World Wide Web into a project in some way. You want to

turn your ideas into reality quickly, and you value tools that help you get the job done. Perhaps you have experience with the popular 8-bit Arduino platform (*http://www.arduino.cc/*), and might even be able to reuse some of your add-on hardware (such as shields and *breakout boards*) originally designed for Arduino.

» Students and hobbyists

  You want your programs to interact with the physical world, using mainstream tools. You are interested in development boards, such as the Netduino Plus, that do not cost an arm and a leg.

» Software developers or their managers

  You need to integrate embedded devices with web services and want to learn the basics quickly. You want to build up an intuition that ranges from overall system architecture to real code. Depending on your prior platform investments, you may be able to use the examples in this book as a starting point for feasibility studies, prototyping, or product development. If you already know .NET, C#, and Visual Studio, you can use the same programming language and tools that you are already familiar with, including the Visual Studio debugger.

  To remain flexible, you want to choose between different boards from different vendors, allowing you to move from inexpensive prototypes to final products without having to change the software platform. To further increase vendor independence, you probably want to use open source platforms, both for hardware and software. To minimize costs, you are interested in a platform that does not require the payment of target royalties, i.e., per-device license costs.

If your background is in the programming of PCs or even more powerful computers, a fair warning: embedded programming for low-cost devices means working with very limited resources. This is in shocking contrast with the World Wide Web, where technologies usually seem to be created with utmost inefficiency as a goal. Embedded programming requires more careful consideration of how resources are used than what is needed for PCs or servers. Embedded platforms only provide small subsets of the functionality of their larger cousins, which may require some inventiveness and work where a desired feature is not available directly. This can be painful if you feel at home with "the more, the better," but it will be fun and rewarding if you see the allure of "small is beautiful."

What You Need to Get Started

This book focuses on the interaction between embedded devices and other computers on the Internet, using standard web protocols. Its examples mostly use basic sensors and actuators, so it is unnecessary to buy much additional hardware besides an inexpensive computer board. Here is a list of things you need to run all the examples in this book:

- A Netduino Plus board (*http://www.netduino.com/netduinoplus*)
- A micro USB cable (normal male USB-A plug on PC side, male micro USB-B plug on Netduino Plus side), to be used during development and for supplying power
- An Ethernet router with one Ethernet port available for your Netduino Plus
- An Internet connection to your Ethernet router
- An Ethernet cable for the communication between Netduino Plus and the Ethernet router
- A potentiometer with a resistance of about 100 kilohm and through-hole connectors
- A Windows XP/Vista/7 PC, 32 bit or 64 bit, for the free Visual Studio Express 2010 development environment (alternatively, you may use Windows in a virtual machine on Mac OS X or Linux, or you may use the Mono toolchain on Linux or Mac OS X)

NOTE: There are several sources where you can buy the hardware components mentioned above, assuming you already have a router with an Internet connection:

- Maker SHED (*http://www.makershed.com/*)
  - Netduino Plus, part number MKND02
  - Potentiometer, part number JM2118791
- SparkFun (*http://www.sparkfun.com/*)
  - Netduino Plus, part number DEV-10186

- Micro USB cable, part number CAB-10215 (included with Netduinos for a limited time)
- Ethernet cable, part number CAB-08916
- Potentiometer, part number COM-09806

For more sources in the U.S. and in other world regions, please see *http://www.netduino.com/buy/?pn=netduinoplus*.

It is also possible to add further sensors and actuators.

Conventions Used in This Book

The following typographical conventions are used in this book:

- *Italic*

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

- `Constant width`

  Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, data types, statements, and keywords.

- ***`Constant width bold`***

  Shows commands or other text that should be typed literally by the user.

- *`Constant width italic`*

  Shows text that should be replaced with user-supplied values or by values determined by context.

NOTE: This style signifies a tip, suggestion, or general note.

Using Code Examples

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

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Getting Started with the Internet of Things*, by Cuno Pfister. Copyright 2011 Cuno Pfister, 978-1-4493-9357-1."

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

How to Contact Us

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

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

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

http://oreilly.com/catalog/0636920013037

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

bookquestions@oreilly.com

For more information about our books, conferences, Resource Centers, and the O'Reilly Network, see our website at:

http://oreilly.com

Safari® Books Online

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

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

O'Reilly Media has uploaded this book to the Safari Books Online service. To have full digital access to this book and others on similar topics from O'Reilly and other publishers, sign up for free at *http://my.safaribooksonline.com*.

Acknowledgments

My thanks go to Brian Jepson, Mike Loukides, and Jon Udell, who made it possible to develop this mere idea into an O'Reilly book. It was courageous of them to take on a book that uses a little-known software platform, bets on a hardware platform not in existence at that time, and addresses a field that is only now emerging. Brian not only edited and contributed to the text, he also tried out all examples and worked hard on making it possible to use Mac OS X and Linux as development platforms.

I would like to thank my colleagues at Oberon microsystems for their support during the gestation of this book. Marc Frei and Thomas Amberg particularly deserve credit for helping me with many discussions, feedback, and useful code snippets. Their experience was invaluable, and I greatly enjoyed learning from them. Marc's deep understanding of REST architecture principles and its implementation for small devices was crucial to me, as was Thomas's insistence on "keeping it simple" and his enthusiasm for maker communities like those of Arduino and Netduino. Both showed amazing patience whenever I misused them as sounding boards and guinea pigs. I could always rely on Beat Heeb for hardware and firmware questions, thanks to his incredible engineering know-how, including his experience porting the .NET Micro Framework to several different processor architectures.

Corey Kosak's feedback made me change the book's structure massively when most of it was already out as a Rough Cut. This was painful, but the book's quality benefited greatly as a result.

I have profited from additional feedback by the following people: Chris Walker, Ben Pirt, Clemens Szyperski, Colin Miller, and Szymon Kobalczyk. I am profoundly grateful because their suggestions definitely improved the book.

The book wouldn't have been possible without the Netduino Plus, and Chris Walker's help in the early days when there were only a handful of prototype boards. Whenever I had a problem, he responded quickly, competently, and constructively. I have no idea when he finds time to sleep.

Last but not least, many thanks go to the team at Microsoft—in particular Lorenzo Tessiore and Colin Miller—for creating the .NET Micro Framework in the first place. Their sheer tenacity to carry on over the years is admirable, especially that they succeeded in turning the platform into a true open source product with no strings attached.

6/Hello Pachube

In this chapter, I will show a basic HTTP client, `HelloPachube`, that pushes samples to Pachube, as shown in Figure 6-1.

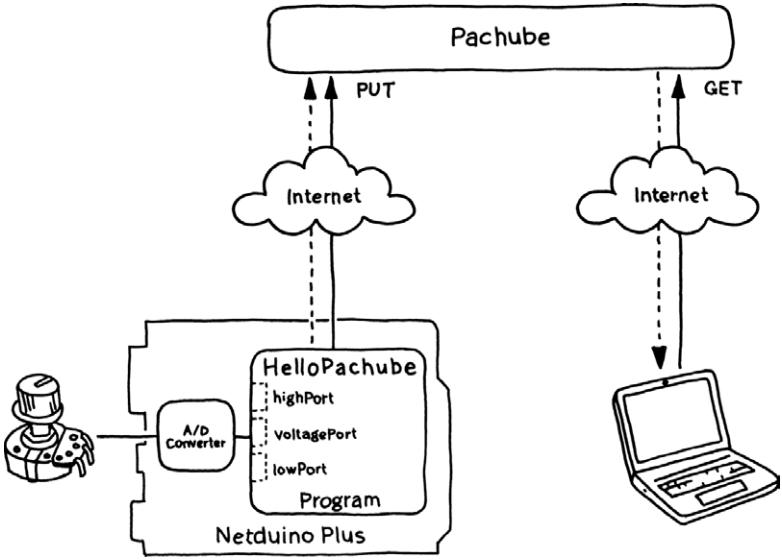


Figure 6-1. Architecture of HelloPachube

`HelloPachube` runs on the Netduino Plus and sends measurements to the Pachube web service by issuing HTTP PUT requests. The user, through his web browser, sends HTTP GET requests to Pachube to retrieve feed entries. The data flow originates in the device, goes up to Pachube, and continues from there to the user.

Setting Up the Network Configuration

Before you can run such a client, you need to make sure that your Netduino Plus board has access to the Internet—i.e., it can send request messages to any server visible on the Internet. I assume that your

Netduino Plus is connected to the Internet via a router and a cable or DSL modem (Figure 6-2).[1] This means that you have a local area network to which both the board and your development PC are connected. During development and debugging, the PC and Netduino Plus are directly connected via a USB cable as well.

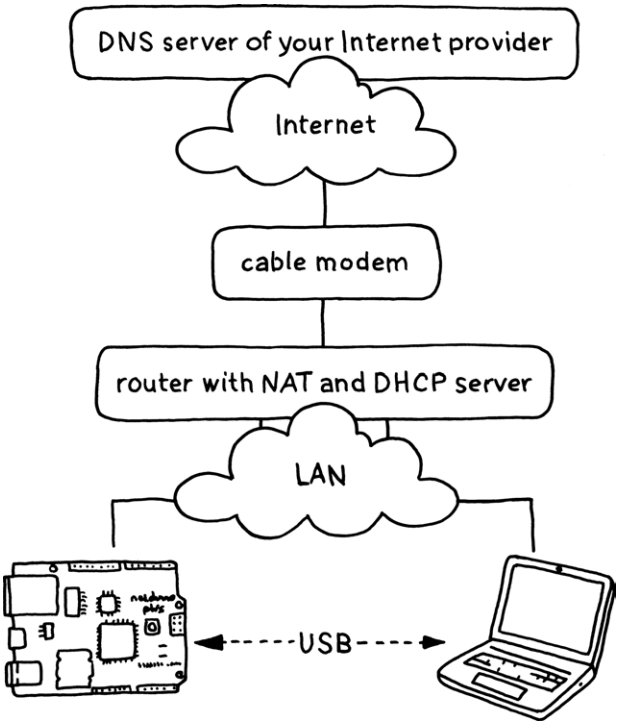


Figure 6-2. Connection of board to the Internet

Internet Addresses

A router typically implements the *Dynamic Host Configuration Protocol* (DHCP). This protocol allows your development PC, your Netduino Plus, and other devices to automatically obtain *Internet addresses* (e.g., `192.168.0.3` for the PC, and `192.168.0.4` for the

[1] Sometimes a cable modem already includes a router in the same box.

Netduino Plus). The Internet protocols rely on Internet addresses for routing messages between clients and servers.

If your Netduino Plus obtains its Internet address automatically via DHCP, it typically gets an Internet address in one of these reserved address ranges:

```
192.168.xxx.xxx
172.16.xxx.xxx
10.xxx.xxx.xxx
```

where *xxx* lies between `0` and `255`. Public Internet servers never use these reserved addresses. They are unique only within a given local area network, not worldwide like other Internet addresses. For example, there are thousands of computers with the *private address* `192.168.1.100`. This is not a problem as long as your device is only a client, but it can be a problem for devices used as servers, as we will see in Part III.

To implement such a multiplexing of Internet addresses, a router has to perform *network address translation* (NAT). This hides the private Internet addresses from the Internet by making it appear as though all Internet traffic from the board or from the development PC originated from the router. This provides a certain degree of security because a program on the Internet cannot directly address—and therefore try to connect to—a device hidden behind the router. In addition, it reduces the number of Internet addresses that must be visible globally, which is important because the common four-byte IPv4 Internet addresses will basically be used up by the time this book comes out.

A client program can directly use an Internet address to connect to a server on the Internet—e.g., the address `173.203.98.29` to connect to a Pachube server. Since such Internet addresses are not very convenient, you can alternatively use a *domain name* for addressing a host. In the above example, the domain name is `pachube.com`. Domain names are registered with the Internet's *domain name system* (DNS). The domain name system allows for looking up domain names, much in the same way as a phone book is used for looking up names (except instead of finding phone numbers, the domain name system returns Internet addresses). A domain name lookup is simply another request over the Internet, e.g., to a DNS server of your Internet service provider.

The MFDeploy Tool

Before you can use your Netduino Plus on the network, you need to check its network settings and configure it if necessary. In particular, you should make sure that DHCP is switched on and that the correct *MAC address* of the board is set. The MAC address is a unique six-byte identifier, typically written like this:[2]

```
3c-8a-4a-00-00-07
```

To check or modify the network configuration, use the tool MFDeploy, which is provided as part of the Microsoft .NET Micro Framework SDK. To find it, click Start→All Programs→Microsoft .Net Micro Framework 4.1→Tools and run MFDeploy.exe. Another way to find it is to look in the directory:

```
C:\Program Files\Microsoft .NET Micro Framework\v4.1\Tools\
MFDeploy.exe
```

(On a 64-bit operating system, the first folder will be *Program Files (x86)*.)

Now, perform the following steps:

1. Start *MFDeploy.exe*. The dialog box .NET Micro Framework Deployment Tool opens.

2. In the leftmost Device list box, change the selection from Serial to USB.

3. Plug your Netduino Plus USB cable into your development PC. In the rightmost Device list box, the name NetduinoPlus_NetduinoPlus should appear.

4. Click on the Ping button to make sure the device responds. As result, the large text box should now show "Pinging... TinyCLR".

5. In the Target menu, select Configuration→Network. The Network Configuration dialog box opens.

6. If it isn't checked already, click on the DHCP checkbox to enable automatic configuration of most network parameters.

[2] You will find the MAC address of your Netduino Plus on the sticker at the bottom of the board.

7. If it isn't configured yet, enter your board's MAC address. This is the only parameter you need to provide. You can leave the DNS Primary Address and the DNS Secondary Address at `0.0.0.0`, as shown in Figure 6-3.

8. Click the Update button.

9. Reboot your Netduino Plus. It should now automatically obtain the missing network parameters from your router. To make sure that the Netduino Plus reboots, I usually perform a complete power-off/power-on cycle by briefly unplugging and reinserting the USB cable from the PC. After such a power cycle, you have five seconds to deploy a new program; otherwise, the most recently deployed program is restarted automatically.

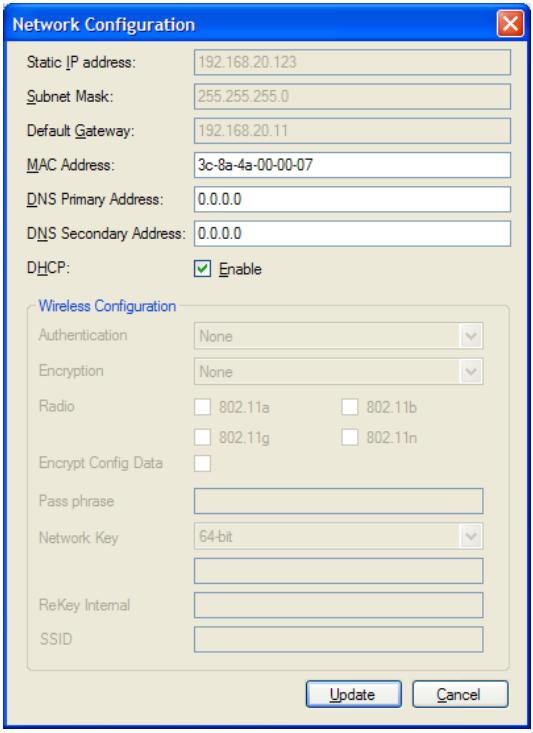


Figure 6-3. Network Configuration in MFDeploy

To check whether the configuration works correctly, run the `HelloPachube` client program described next.

HelloPachube

Now that your Netduino Plus is ready to access the Internet, we can look at a first version of a Pachube client. Its source code is given in Example 6-1.

Example 6-1. HelloPachube

```
using System;
using System.Threading;
using Gsiot.PachubeClient;
using Microsoft.SPOT;
using Microsoft.SPOT.Hardware;
using SecretLabs.NETMF.Hardware;
using SecretLabs.NETMF.Hardware.NetduinoPlus;

public class HelloPachube
{
    public static void Main()
    {
        const string apiKey = "your Pachube API key";
        const string feedId = "your Pachube feed id";
        const int samplingPeriod = 20000;   // 20 seconds

        const double maxVoltage = 3.3;
        const int maxAdcValue = 1023;

        var voltagePort = new AnalogInput(Pins.GPIO_PIN_A1);
        var lowPort = new OutputPort(Pins.GPIO_PIN_A0, false);
        var highPort = new OutputPort(Pins.GPIO_PIN_A2, true);

        while (true)
        {
            WaitUntilNextPeriod(samplingPeriod);
            int rawValue = voltagePort.Read();
            double value = (rawValue * maxVoltage) / maxAdcValue;
            string sample = "voltage," + value.ToString("f");
            Debug.Print("new message: " + sample);
            PachubeClient.Send(apiKey, feedId, sample);
        }
    }
```

```
    static void WaitUntilNextPeriod(int period)
    {
        long now = DateTime.Now.Ticks / TimeSpan.TicksPerMillisecond;
        var offset = (int)(now % period);
        int delay = period - offset;
        Debug.Print("sleep for " + delay + " ms\r\n");
        Thread.Sleep(delay);
    }
}
```

To run the program, follow these steps:

1. Make sure that your Netduino Plus is connected to your Ethernet router and that it is correctly configured for network access (see the previous section).

2. If you haven't done so already, download the Visual Studio project `Gsiot.PachubeClient` from *http://www.gsiot.info/download/*, unzip it, and put it into the *Visual Studio 2010\Projects* directory.

3. Create a new Visual Studio project (using the Netduino Plus template) and name it HelloPachube. Replace the contents of *Program.cs* with the code from Example 6-1.

4. You must replace the strings for `apiKey` and `feedId` so they match your Pachube API key and feed ID.

5. Right-click on References in the Solution Explorer. Select Add→ New Reference. In the Add Reference dialog box, click on the Browse tab. In the directory hierarchy, go up two steps to directory *Project*. In the directory *Gsiot.PachubeClient*, open the subdirectory *Gsiot.PachubeClient* (yes, the same name again). In this directory, open the *bin* subdirectory. From there, open the *Release* subdirectory. In this subdirectory, select the *Gsiot.PachubeClient.dll* file. Click the OK button. You have now added the assembly *Projects\Gsiot.PachubeClient\Gsiot.PachubeClient\bin\Release\Gsiot.PachubeClient.dll*.

Now you're ready to test it: build the project and deploy it to your Netduino Plus, as described in the section "Deploying to the Device" in Chapter 1.

NOTE: In the simplest case, one C# namespace is translated into exactly one .NET assembly (stored in a DLL), which is the binary form of .NET code. For the .NET Micro Framework, a built-in postprocessor tool translates *.dll* assembly files into *.pe* files, which are a more compact representation of the same code. These are the files that get deployed to the Netduino Plus.

Viewing the Results

After `HelloPachube` has started, you'll see something like the following in Visual Studio's Output window:

```
sleep for 19069 ms

The thread '<No Name>' (0x3) has exited with code 0 (0x0).
new message: voltage,0.06
time: 01/01/2009 02:16:40
memory available: 20136
Status code: 200
sleep for 19371 ms

The thread '<No Name>' (0x4) has exited with code 0 (0x0).
new message: voltage,0.06
time: 01/01/2009 02:17:00
memory available: 20136
Status code: 200
sleep for 19210 ms

The thread '<No Name>' (0x5) has exited with code 0 (0x0).
new message: voltage,1.52
time: 01/01/2009 02:17:20
memory available: 20136
Status code: 200
sleep for 19369 ms

The thread '<No Name>' (0x6) has exited with code 0 (0x0).…
```

Because a Netduino Plus has no battery-backed real-time clock, its clock is started anew whenever you reboot the device. Upon rebooting, the initial time is the start of January 1, 2009.

Twenty seconds pass between two consecutive samples; roughly 19 of them are spent sleeping. You can see that the samples were successfully

sent to Pachube because the returned status code is 200, which is the OK status code of HTTP.

To verify that the samples have indeed arrived at Pachube, type the following URI into your web browser, replacing *your Pachube feed id* with your feed ID:

```
http://www.pachube.com/feeds/your Pachube feed id
```

You should now see that the status of your feed is marked as `currently: live`. This means that the most recent sample is not older than 15 minutes; otherwise, the status `currently: frozen` would be shown.

NOTE: If you don't see this output, make sure that the Netduino Plus is connected via Ethernet cable to a router, and via USB cable to your development PC. Use MFDeploy to check whether DHCP is enabled and the MAC address is set. Check whether the example correctly builds and whether its properties are set up to deploy to the device via USB.

To see a graphical representation of the most recent samples, view the feed's web page, look at the graph there, and click the label "last hour".

How It Works

The initialization of the HelloPachube `Main` method starts with two Pachube-related constants: your Pachube API key (`apiKey`) and the ID of the feed to which you want to publish your samples (`feedId`). After that, there are a few other constants and variables that are set:

» Specifying how often to send updates

  First comes the timing-related constant `samplingPeriod`. The goal for the example is to sample and publish a new observation at regular intervals, namely once every `samplingPeriod`, which is given in milliseconds (20,000 milliseconds is 20 seconds).

  To publish a sample, send a web request and wait for its response. The time for such a complete *round-trip* consists of the time it takes for the request to travel to the server, for the server to create a response, and for the response to travel back to the client.

NOTE: The speed of the round-trip depends mainly on five factors: the distance between client and server, the current traffic on the Internet, the performance of the server, the current load of the server, and the amount of data transferred. Typical numbers range from about 50 milliseconds for round-trips to servers close to the client, to well over 1,000 milliseconds for round-trips across continents or to overtaxed servers (even for short messages). Since they depend on the Internet's current traffic, the times for subsequent round-trips from the same client to the same server can vary.

If you use a slower connection than Ethernet, this can also affect round-trip times. For example, if you dropped a Netduino out in the woods with a cheap 2G GSM module, it would probably spend most of the 20 seconds doing the round-trip.

- Setting up the voltage reader

  The `voltagePort` object and related variables and constants are set up, as you saw in Chapter 3. They are used for reading voltage values from an attached potentiometer.

After the variables and constants are initialized, a `while` loop controls what happens from then on. This *main loop* will run until you turn off the Netduino Plus.

The main loop does basically three things:

- Sleeps until the next sample is due, using the helper method `WaitUntilNextPeriod`, which I will discuss in the next section.
- Creates the sample by reading the voltage port.
- Sends the value to Pachube using `PachubeClient.Send`. This method takes the Pachube API key, your feed ID, plus the sample data, and sends them to Pachube in a suitable PUT request message. It then receives the response message and prints the response's status code to the debug console.

To use the `Gsiot.PachubeClient` for sending requests to Pachube in a "fire and forget" manner, you don't need to know more than this. However, if you want to know how the library actually works, how you could modify it, or how you could create a similar library, you need to understand more about how to send HTTP request messages and receive HTTP response messages. This is the topic of Chapter 7.

The WaitUntilNextPeriod Method

In this example, samples should be taken at highly regular intervals. To do this, you can use the `WaitUntilNextPeriod` helper method, which you can reuse in similar programs later on. The following text explains the method in some detail. You can skip the explanation if you just want to go ahead and use the method.

After each sample is sent, the program needs to sleep until the next period starts. How can this delay be calculated with precision when we don't know in advance exactly how long it will take to send a request and receive its response?

This example starts a new period every 20 seconds. (Free Pachube accounts don't allow updates more often than every 12 seconds.) Assume the following:

- You last took a reading at 09:32:40.
- After the time it took you to send a message and receive the response, it is now 09:32:46.
- You want to send the next message (and start a new period) at 09:33:00.

The delay then can be calculated as the difference between the length of the period (20 seconds) and the offset, where the offset indicates how far you are into the current period. The offset is calculated as the current time (i.e., now) modulo the period. In the example shown in Figure 6-4, the offset is six seconds; therefore, the delay is 14 seconds.

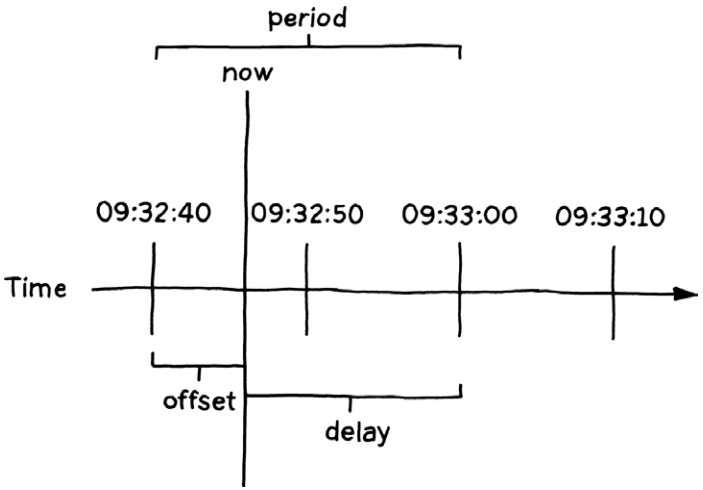


Figure 6-4. Calculating the delay until the start of the next period

The property `DateTime.Now.Ticks` gives the current time[3] in *ticks*, which in .NET is a time at a resolution of 100 nanoseconds. Dividing ticks by 10,000 (`TimeSpan.TicksPerMillisecond`) yields the same time in milliseconds, albeit less precisely. This requires a 64-bit `long` integer type. To calculate the modulus, use the `%` operator of C#. Because the result of a modulus operation is always smaller than the operand, in this case `period`, it can be safely cast to a 32-bit integer using the `(int)` cast.

NOTE: The modulo operator, `%`, computes the remainder of a division. For example, the division of 7 by 2 yields 3. Computing "backwards" by multiplying the result 3 by the divisor 2, we get 6. The difference between 6 and the dividend (7) is the remainder—in this case, 1.

`WaitUntilNextPeriod` ensures that sampling starts at highly regular intervals. It is robust even in cases where an iteration takes longer than its period allows for. This might occur if something unexpected happens, such as an exception that takes an inordinately long time to be sent to the debugger. This may result in one or several periods being skipped—but the next one starts at a correct period boundary anyway.

[3] On a Netduino, this is the time since the device has booted. It has no battery-backed real-time clock that keeps track of time when it isn't powered.

Casting Values

C# provides several integer types, which differ in the number ranges that they encompass and in the bits used for storing them. The larger the number range, the more bits are needed. The `int` type supports numbers in the range from –2,147,483,648 to 2,147,483,647 and requires 32 bits (four bytes). The `long` type supports numbers in the range from –9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 and requires 64 bits (eight bytes). Any `int` variable `i` fits in a `long` variable `l`, so the assignment `l = i` always works. However, the other direction does not always work: most `long` numbers do not fit in an `int` variable. Therefore, C# requires using a type cast, `i = (int)l`, to make it obvious that this danger exists here. If `l` is too large when it is assigned to `i`, `i` will be assigned garbage. So when the compiler requires such a type cast, it is a good idea to think about whether you can really be sure that the current value of `l` fits into `i`.

What Netduino Said to Pachube

To see that there is no magic involved in HTTP requests, let's look at the data actually transferred to the Pachube server during a request:

```
PUT /v2/feeds/fid.csv HTTP/1.1\r\n
Host: api.pachube.com\r\n
X-PachubeApiKey: your Pachube API key is here\r\n
Content-Type: text/csv\r\n
Content-Length: 12\r\n
\r\n
voltage,1.52
```

This is the text sent over the Internet to Pachube! At least that's what is sent if the measured voltage is 1.52.

An HTTP request consists of one *request line*, followed by a number of *header lines*, followed by an empty line, and optionally followed by a *message body* (i.e., the message's *content*).

The request line starts with the HTTP method: PUT, GET, etc. After a blank, the request URI indicates the resource to be accessed. After another blank, the HTTP version is given, which is usually version 1.1 these days. The request line is terminated by a carriage-return byte followed by a newline byte.

NOTE: `\r\n` stands for the two bytes `CR` and `LF` (carriage return and line feed, respectively). If you were to look at the actual text of the request, they would not be visible.

HTTP defines a number of headers, both for requests and responses. For requests, the `Host` header is particularly important because it defines to which computer the request is sent—in this case, to `api.pachube.com`. If you take this host and the request URI in the request line (here, it's `/v2/feeds/`*`fid`*`.csv`), you can construct the absolute URI of the resource accessed by this PUT request:

```
http://api.pachube.com/v2/feeds/fid.csv
```

Unlike the URIs that we have seen in Chapter 5, which have been URIs for consumers of Pachube feeds, this is a URI for producers that send measurements to Pachube.

Different applications may use very different sets of headers. For our purposes, the most important headers are `Host` (for requests only) and `Content-Length` and `Content-Type` (for both requests and responses). Applications may define their own headers, like the `X-PachubeApiKey` above. The order of HTTP headers is not significant, as every possible ordering is correct.

The message body consists of exactly the 12 bytes `voltage,1.52` here, has no terminating characters, and is separated from the last header by an empty line.

NOTE: To find out what exactly your client is sending, you may use a simple test server (such a server is given in Appendix A). To make `Hello-Pachube` send its requests to a test server running on your PC, change the constant `baseUri` in `Gsiot.PachubeClient` so that it points to your server.

What Pachube Said to Netduino

An HTTP response from Pachube may look like this:

```
HTTP/1.1 200 OK\r\n
Server: nginx/0.7.65\r\n
Date: Mon, 07 Feb 2011 13:36:55 GMT\r\n
Content-Type: text/plain; charset=utf-8\r\n
Connection: keep-alive\r\n
Set-Cookie: _pachube_app_session=BAh7BjoPc2Vzc2lvbl9…;\r\n
Cache-Control: max-age=0\r\n
Content-Length: 1\r\n
Age: 0\r\n
Vary: Accept-Encoding\r\n
```

In this response, the first line, known as the *status line*, is the most important. HTTP defines a number of status codes; status code 200 means that the request was handled successfully. (The most important status codes are given in Chapter 10.) The status code is located between the HTTP version and a plain-text version of the status code. The text version of the status code is optional—you neither need to generate nor interpret it. It is merely a convenience for human readers of HTTP interactions.

Responses may contain many headers, as you can see from this example. Fortunately, you can usually ignore almost all of them. Nevertheless, let's take a look at the headers in the response:

» `Server`

  Indicates the web server software that Pachube uses.

» `Date`

  Indicates the time when Pachube has sent the response.

» `Content-Type: text/plain; charset=utf-8`

  Indicates the format of the Pachube response. In this case, it is plain text encoded in UTF8 (the most common encoding of Unicode characters).

NOTE: Actually, with many web services, the response to a successful PUT request has an empty message body. The server is allowed to return a response body, though.

» `Connection: keep-alive`

  Is a relic from HTTP 1.0 (an early version of the HTTP specification). Originally, a new TCP/IP connection was opened for every request and then closed after the request. Because opening a connection incurs a considerable overhead, it is better to keep a connection open if requests are sent to the same server every couple of seconds. The `keep-alive` value was added to indicate this desire. It is not relevant anymore because most servers and clients today support HTTP 1.1, where connections are kept alive by default. However, if for any reason a client or a server wants to close a connection after a message exchange, it can signal this to the other party by including the `Connection: close` header.

NOTE: A connection may also be closed even while request or response messages are being exchanged—e.g., if someone tripped over your Ethernet cable and it was yanked out. This means that closed connections must be reopened if necessary, lost messages may have to be re-sent, and clients and servers must be programmed in a way that they do not misbehave—even if a connection is closed.

» `Set-Cookie`

  Indicates a cookie (some text that the server sends a client to store, and which the client will send to the server in future requests) with a session identifier. You can ignore cookies because they are not needed for our examples.

» `Cache-Control: max-age=0`

  Is intended for managing caches between client and server. It indicates that this response must not be cached.

» `Content-Length: 1`

  Indicates that the response message body consists of one byte.

» `Age: 0`

  Is an estimate (in seconds) of the time it has taken to produce and transmit the response. It is a header produced by some intermediary cache between server and client. You can ignore it.

» `Vary: Accept-Encoding`

  Tells the client that it may send an `Accept-Encoding` header along with GET requests, in order to ask for different representations of the resource. As we have seen in Chapter 5, Pachube supports several formats for samples: `csv`, `json`, `png`, etc. However, you won't need the `Accept-Encoding` header in the examples of this book. Instead, you can pass the desired format as part of the URI, e.g., `http://api.pachube.com/v2/feeds/256.csv`.

The message body, after the last `CR LF` (empty line), consists of exactly one blank character. It seems a bit strange that it is not completely empty in the case of Pachube, but you can usually ignore the message body of a PUT response anyway.

HTTP requests and responses are not complicated. Any device capable of supporting TCP/IP is able to send data to Pachube or to similar services.

Index

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

About the Author

Dr. Cuno Pfister studied computer science at the Swiss Federal Institute of Technology in Zürich (ETH Zürich). His PhD thesis supervisor was Prof. Niklaus Wirth, the designer of the Pascal, Modula-2, and Oberon programming languages. Dr. Pfister is the Managing Director of Oberon microsystems, Inc., which has worked on various projects related to the Internet of Things, from mobile solutions to a large hydropower-plant monitoring system with 10,000 sensors.

Colophon

The cover, heading, and body font is BentonSans, and the code font is Bitstreams Vera Sans Mono.