A
Console Applications

This appendix shows the basic structure of console applications, as used throughout the examples in this book.

Creating the solution and project files

To create a new solution in Xamarin, go to the **File** menu, select **New**, and then select **Solution**. Select **Blank Solution** from the **Other** category or perhaps **Console Project** from the **C#** category. The latter creates a console project for you at the same time. Don't forget to enter a name for the solution and the location to store the files. To add a new project to the solution, right-click on the solution in the **Solution** panel to the left in the IDE and select **Add** and then **Add New Project**. You can also add the `Clayster.Library.IoT` and `Clayster.Library.RaspberyPi` projects by selecting **Add Existing Project** from the same pop-up menu.

Once the project is created, you need to add project references to the project. References tell the compiler that these projects are required by the running application. To add references to the project, right-click on the `References` folder of the newly created `Sensor` project in the **Solution** panel. From the context menu that appears, select **Edit References**. In the **References** dialog that appears, you need to add three types of libraries. First, you need to add the `System.Xml` and `System.Drawing` .NET libraries to the project. This is done from the **Packages** page in the **References** dialog. These two libraries allow you to work with XML and images in a simple way. Then, you need to add references to the `Clayster.Library.IoT` and `Clayster.Library.RaspberyPi` libraries, to which source code is provided, if these are added to the solution. This is done in the **Projects** tab of the same dialog. In this tab, you will see all the projects in your solution. Lastly, you need to add references to the remaining Clayster libraries. This is done in the **.NET Assembly** tab in the same dialog. Navigate to the folder with the downloaded libraries and add references to the corresponding `.dll` files to the project.

Basic application structure

When creating a new console project in Xamarin, the main program file will be named `Program.cs` and will look as follows:

```
using System;
namespace Sensor
{
  class MainClass
  {
    public static void Main (string[] args)
    {
      Console.WriteLine ("Hello World!");
    }
  }
}
```

Logging events

All projects in this book will use the following setup, and we describe it only here for brevity. First, we will add the following `using` statements, since we will not only use multiple threads but also the sleep function and event logging:

```
using System.Threading;
using Clayster.Library.EventLog;
using Clayster.Library.EventLog.EventSinks.Misc;
```

The event logging architecture allows for any number of event sinks to be registered. Event sinks can be used to analyze the event flow, store events, or send events to the network somewhere. If event logging is done properly when building applications, it's easy at a later stage to add more advanced event logging capabilities, for instance, sending events from things to a central repository for monitoring and analysis. For our purposes, it is sufficient at this point to only output events to the terminal window. For this reason, we will add the following code to the top of the `Main()` method:

```
Log.Register (new ConsoleOutEventLog (80));
Log.Information ("Initializing application...");
```

Terminating gracefully

We then register an event handler that will be executed if the user presses *CTRL+C* in the terminal window when executing the application. Until this key combination is pressed, the `Executing` variable will remain `true`, as shown in the following case:

```
bool Executing = true;
Console.CancelKeyPress +=
  (object sender, ConsoleCancelEventArgs e) =>
  {
    e.Cancel = true;
    Executing = false;
  };
```

By adding the previous event handler, we can implement a graceful shutdown of our console application, as follows:

```
Log.Information ("Application started...");
try
{
  while (Executing)
  {
    System.Threading.Thread.Sleep (1000);
  }
}
catch (Exception ex)
{
  Log.Exception (ex);
}
finally
{
  Log.Information ("Terminating application.");
  Log.Flush ();
  Log.Terminate ();
}
```

Note that any unexpected exceptions should always be caught and sent to the event log. This makes it easier to detect errors in the code. Furthermore, we need to terminate the event log properly by using the `Terminate` method, or the console application will not be terminated since there are active threads still running.

Compiling and deploying the project

When compiling the application, executable files will be generated and stored in the `bin/Debug` folder under the project folder. Files with the extension `.dll` are executable library files. The file with the `.exe` extension is the executable file. Files with the `.pdb` extension are debug files. If they are downloaded, remote debugging is possible and stack traces will contain line number information to help locate errors quickly.

To deploy files to a Raspberry Pi, several methods are available. The method used in our examples includes using a command-line version of the **Secure Copy** (**SCP**) protocol, which copies files using the **Secure Shell** (**SSH**) protocol, a protocol used by Linux for secure terminal connections to the device. A command-line version of SCP called `PSCP.exe` is available in PUTTY, the terminal application we used when creating the applications.

To simplify automatic deployment, each project has a file called `CopyToRaspberryPi.bat` that copies relevant files to the corresponding Raspberry Pi. To automatically deploy newly compiled code, right-click on the project in Xamarin and select **Options**. In the **Options** dialog, go to **Build and Custom Commands**. Choose **After Build** and select and execute the `CopyToRaspberryPi.bat` command in the `${ProjectDir}` working directory. Now, the batch file will execute every time the project has been successfully built, copying all files to the corresponding Raspberry Pi. To make deployment quicker, files seldom changed can be commented out. The following line shows an example of how the command-line syntax of deploying a file will look on a Windows machine:

```
"c:\Program Files (x86)\PuTTY\pscp.exe" -pw raspberry
bin/Debug/Sensor.exe pi@192.168.0.29:
```

To execute the file on a Raspberry Pi, simply execute the following in a terminal window:

```
$ sudo mono Sensor.exe
```

Since `Sensor.exe` is a .NET application, it needs to run within the Mono virtual machine. To make sure the application has full access rights, which is important to access GPIO later, super user access rights are required.

Making the application run at system startup

When the sensor is done, we might want to configure our Raspberry Pi to automatically run the application when it boots. This way, it will always start when the Raspberry Pi is powered up. To do this, open a terminal window to the Raspberry Pi and edit the `/etc/rc.local` file as follows:

```
$ sudo nano /etc/rc.local
```

Before the `exit` statement, we add the following:

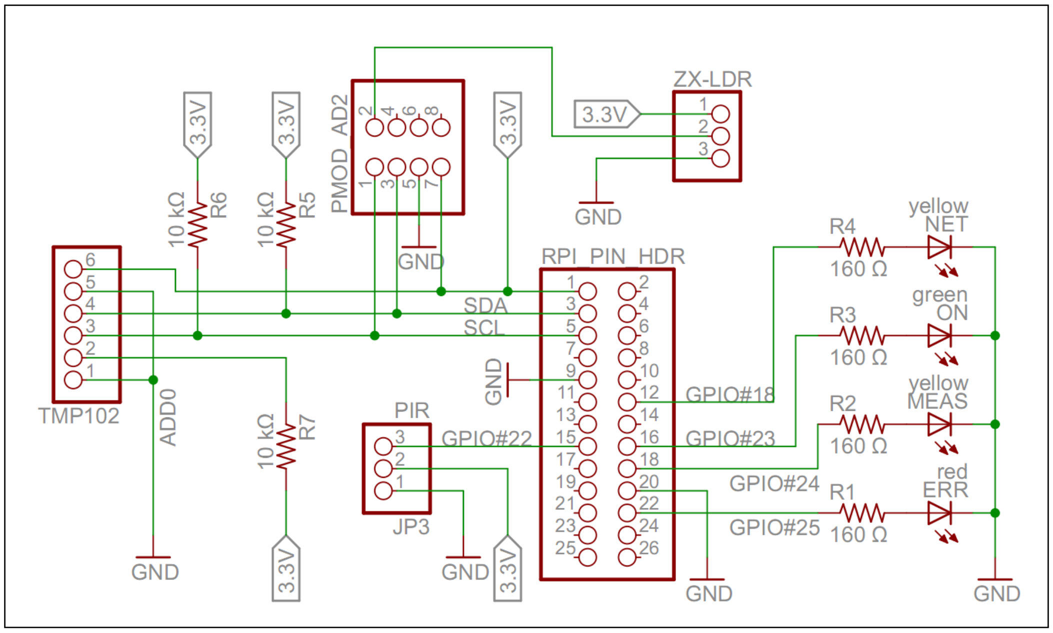
```
cd /
cd home
cd pi
mono Sensor.exe > /dev/null &
```

We can now exit, save the file, and reboot the Raspberry Pi. After a few moments, the LEDs on our prototype board will indicate that the sensor application is up and running. Navigating to the sensor in a browser will also confirm the sensor is alive and well.

To update the application at a later stage, you need to kill the Mono process first, update the application, and test it; then, when you are satisfied, reboot the device and the application will automatically start again, using the new version of the code.

B
Sampling and History

Performing basic sampling and keeping a historical record of sampled values is the basic function of any sensor. Sensors are an important aspect of Internet of Things. This appendix shows how sampling and historical record keeping is done in the sensor project published in this book. You start by creating a project, as described in *Appendix A*, *Console Applications* and then follow it up with the instructions in this appendix. Here, we will start by interfacing our hardware, configuring it, preparing the code with the basic data structures, and then sampling values sensed by the hardware. The circuit diagram for our prototype board, as described in *Chapter 1*, *Preparing our IoT Projects*, is as follows:



Interfacing the hardware

We start by adding code to interface the hardware on our prototype board. The actual interface with GPIO is done using the `Clayster.Library.RaspberryPi` library, for which you'll have the source code available. We first add the following references to the corresponding namespaces:

```
using Clayster.Library.RaspberryPi;
using Clayster.Library.RaspberryPi.Devices.Temperature;
using Clayster.Library.RaspberryPi.Devices.ADC;
```

The `RaspberryPi` namespace contains generic GPIO classes, while the `Devices` subnamespace contains classes for communication with specific devices. We then create the following private static members, one `DigitalOutput` class for each one of the LEDs:

```
private static DigitalOutput executionLed =
  new DigitalOutput (23, true);
private static DigitalOutput measurementLed =
  new DigitalOutput (24, false);
private static DigitalOutput errorLed =
  new DigitalOutput (25, false);
private static DigitalOutput networkLed =
  new DigitalOutput (18, false);
```

We also remove the `Executing` variable defined in *Appendix A*, *Console Applications*, and replace it with `executionLed.Value`. Instead of setting the variable to `true` or `false` respectively, we can also use the `High()` and `Low()` methods. By using this LED instead of an internal variable, we can physically see when the application is running.

The `DigitalOutput` class manages the state of an output GPIO pin. The first parameter is the GPIO pin number it controls and the second parameter is its initial state. We also need to add an object of the `DigitalInput` class for the motion detector on GPIO pin 22, as follows:

```
private static DigitalInput motion = new DigitalInput (22);
```

We then have two sensors connected to an I^{2}C bus that is connected to pins 3, **Serial Clock** (**SCL**), and 2, **Serial Data** (**SDA**). If a Raspberry Pi **R1** is used, these pins have to be changed to pin 1 instead of 3 for SCL and pin 0 instead of 2 for SDA. Reading the component specifications, we deduce that a maximum clock frequency of 400 kHz is allowed. We code these specifications in the following simple statement:

```
private static I2C i2cBus = new I2C (3, 2, 400000);
```

We then add a reference to the Texas Instruments TMP102 sensor, hardwired to address 0, which within the class is converted to I^{2}C address 48 hex, as follows:

```
private static TexasInstrumentsTMP102 tmp102 =
  new TexasInstrumentsTMP102 (0, i2cBus);
```

The Analog/Digital Converter Digilent Pmod AD2 employed uses an Analog Devices AD7991, which also uses I^{2}C to communicate with microcontrollers. It is also hardwired to address 0, which internally in the class is converted to I^{2}C address 28 hex, making it possible to coexist with the temperature sensor. Only one of the A/D channels is used in this example. We add the corresponding interface as follows:

```
private static AD799x adc =
  new AD799x (0, true, false, false, false, i2cBus);
```

Correctly releasing hardware

Hardware attached to GPIO pins are not released by default when an application terminates, as is done with other system resources controlled by the operating system. For this reason, it is very important to always release hardware resources correctly in the application and shut down the application gracefully, regardless of what happens inside the application. For this reason, we call the `Dispose()` method on all hardware resources in the `finally` statement at the end of the `Main` method, which is guaranteed to run. The `Dispose()` method makes sure all resources are released correctly, and any output pins are converted back to passive input pins:

```
executionLed.Dispose ();
measurementLed.Dispose ();
errorLed.Dispose ();
networkLed.Dispose ();
motion.Dispose ();
i2cBus.Dispose ();
```

Internal representation of sensor values

We now have our hardware interfaces in place. We now need to plan how to represent sampled values internally. The motion detector is a digital input. Its internal representation is simply done as a Boolean parameter, as follows:

```
private static bool motionDetected = false;
```

The temperature sensor returns a binary 16-bit value whose most significant byte (8 bits) corresponds to an integer number of degrees centigrade, and the least significant byte (bits) corresponds to fractions of degrees. Negative values are represented using two's complement, which means the 16-bit value should be treated as a simple signed 16-bit integer (`short`) in C#. We will convert this to a `double` datatype for practical reasons that will become clear later. So, our internal representation of the temperature value becomes this:

```
private static double temperatureC;
```

The light sensor, on the other hand, is only a simple analog device with no calibrated physical unit. The AD7991 device will return as a 12-bit unsigned value from 000 to FFF hex. We will convert this to a relative value in percentage, where 0 percent will represent no light and 100 percent maximum light, all as measured by the sensor. Practically, during a bright day or when using a flashlight, the sensor will measure 100 percent. When placing one or two hands on the sensor, it will measure 0 percent. Our internal representation of light density will also be made by a `double` value, as follows:

```
private static double lightPercent;
```

Since access to sensor values will be possible from multiple threads, we will also create a synchronization object, which will be used during the lifetime of the application—except during initialization—to make sure data is always consistent:

```
private static object synchObject = new object ();
```

Averaging to decrease variance

Our application will sample the physical values every second. To avoid jitter in sampled values, we will also use an averaging window that for each sample will calculate the average value of the last ten sampled values. Such an averaging window will reduce the variance in sampled values and remove jitters that often occur when sensors are sampled frequently and differences in sampled values is small. This reduction in variance will appear to also decrease error and provide an additional decimal of accuracy. Even though the method actually reduces sample errors, it does not remove systematic errors that make sensors offset over time. To remove such errors, recalibration of sensors is required. But, by using the average values over the last ten sampled values, the output becomes smoother. And if you're measuring only the change in sampled values, the sensing becomes more accurate and gives an additional decimal of precision.

So, we add the following member variables to the application to be able to calculate the average values of the last ten samples:

```
private static int[] tempAvgWindow = new int[10];
private static int[] lightAvgWindow = new int[10];
private static int sumTemp, temp;
private static int sumLight, light;
private static int avgPos = 0;
```

The `avgPos` variable maintains the position in the averaging windows. The `sum*` parameter contains the sum of all values in the averaging window, and the `temp` and `light` variables contain the most recent sample. Note that the sums are made on integers, which means we do the summation on binary raw values and not floating point values. This removes the possibility that the operation will introduce round off errors over time, which would otherwise be the result if a large amount of floating point additions and subtractions would have been used.

Configuring and initializing the temperature sensor

Before the application can start using the sensors, we need to initialize them correctly. We also need to initialize member variables used for sensing. To initialize the temperature sensor and its sensing variables, we do as follows:

```
try
{
  tmp102.Configure (false,
    TexasInstrumentsTMP102.FaultQueue.ConsecutiveFaults_6,
    TexasInstrumentsTMP102.AlertPolarity.AlertActiveLow,
    TexasInstrumentsTMP102.ThermostatMode.ComparatorMode,
    false, TexasInstrumentsTMP102.ConversionRate.Hz_1, false);

  temp = (short)tmp102.ReadTemperatureRegister ();
  temperatureC = temp / 256.0;

  for (int i = 0; i < 10; i++)
    tempAvgWindow [i] = temp;

  sumTemp = temp * 10;
}
catch (Exception ex)
{
  Log.Exception (ex);
  sumTemp = 0;
  temperatureC = 0;
  errorLed.High ();
}
```

The first statement is TMP102-specific and configures how the device should operate. The first parameter (`false`) disables the one-shot feature, which in practice means the device performs regular sampling. The second parameter states that the sensor should flag for sensor errors only after six consecutive faults have occurred. The third parameter controls the ALERT pin on the temperature sensor saying that it should be active low, meaning it is high in a normal state and is pulled low when an error occurs. The ALERT pin is not used in our application. The fourth parameter configures the sensor to work in normal comparator mode and not interrupt mode. We don't use the sensor's interrupt pin in our application, so we leave it in comparator mode. The fifth parameter tells the sensor to sample the temperature every second. In the sixth parameter, we disable the extended mode, which would give us an extra bit of precision. Normal mode is sufficient for our application.

The rest of the code is easier to understand. The temperature sensor is read, the averaging window is filled with the current value, and the sum register is set accordingly. This assures that the average calculation of the following sample will be calculated correctly. If an exception occurs, as would happen if the temperature sensor cannot be read, the error LED is lit and variables are filled with zeroes.

Configuring and initializing the light sensor

We must now do the same with the light sensor, or better said, with the A/D converter. The only thing that differs is how the hardware is initialized and how the momentary value is calculated:

```
  try
  {
    adc.Configure (true, false, false, false, false, false);

    light = adc.ReadRegistersBinary () [0];
    lightPercent = (100.0 * light) / 0x0fff;

    for (int i = 0; i < 10; i++)
      lightAvgWindow [i] = light;
    sumLight = light * 10;
  }
  catch (Exception ex)
  {
    Log.Exception (ex);
      sumLight = 0;
    lightPercent = 0;
    errorLed.High ();
  }
```

When configuring the AD7991 A/D converter, the first four parameters state which channels are active and which are not. In our example, only the first channel is active. The fifth parameter states that we do not use an external voltage reference connected to one of the input channels, but use the same voltage reference used to power the I^{2}C communication bus. The sixth parameter tells the converter not to bypass existing filters on the I^{2}C SCL and SDA pins.

Setting up the sampling interval

We are now ready to perform the actual sampling. As mentioned previously, sampling will be performed in the application every second. To activate this sampling frequency, we add the following to the main method, just before entering into the main loop:

```
Timer Timer = new Timer (SampleSensorValues, null,
  1000 - DateTime.Now.Millisecond, 1000);
```

This line of code creates a `System.Threading.Timer` object that will call the `SampleSensorValues` method every 1,000 milliseconds (last parameter). The first call will be made on the next even second shift (third parameter). The timer method takes a state object, which we do not need, so we choose to send a `null` value (second parameter). To make sure the timer is disposed correctly when the system terminates, we add the following line to the application's clean-up clause at the end of the main method:

```
Timer.Dispose ();
```

Performing the sampling

First we create the method that will be called by the sample timer created previously. This method takes an object-valued parameter, which will always be `null` in our case. We will light the measurement LED at the beginning of the method and make sure the LED is unlit at the end. The event handler is also secured using try-catch-finally to make sure unhandled exceptions do not make the entire application fail. The actual sampling will be done within the `lock` statement that is there to make sure access to the sample parameters can only be done from one thread at a time. If sampling goes well, the error LED is unlit (if lit):

```
private static void SampleSensorValues (object State)
{
  measurementLed.High ();
  try
  {
    lock (synchObject)
    {
    }
    errorLed.Low ();
  }
  catch (Exception)
  {
    errorLed.High ();
  }
  finally
  {
    measurementLed.Low ();
  }
}
```

Within the `lock` statement, we can now start our sampling. We begin by reading the current raw values from the temperature and light sensors:

```
temp = (short)tmp102.ReadTemperatureRegister ();
light = adc.ReadRegistersBinary () [0];
```

We then subtract the oldest values available in the averaging window from the corresponding sum variables, replace the oldest values with the newest, and add these values to the sum registers:

```
sumTemp -= tempAvgWindow [avgPos];
sumLight -= lightAvgWindow [avgPos];
tempAvgWindow [avgPos] = temp;
lightAvgWindow [avgPos] = light;
```

```
sumTemp += temp;
sumLight += light;
```

We then update the momentary value registers by calculating the average value of the latest ten measurements. We also make sure to move the averaging window position to the next oldest value, which after the current operation is the oldest:

```
temperatureC = (sumTemp * 0.1 / 256.0);
lightPercent = (100.0 * 0.1 * sumLight) / 0x0fff;
avgPos = (avgPos + 1) % 10;
```

We also make sure to update the motion detector variable with its current status:

```
motionDetected = motion.Value;
```

Historical records

One of the advantages of using a plug computer is that it is easy to store and process historical data. We will take advantage of this fact and store historical values each minute, hour, day, and month. But instead of storing current values at even time intervals, which might be misleading, averaging will be performed for the entire time interval of each corresponding period. In this case, averaging will not use windows since the average is only required at the end of a period. For the binary motion value, we will consider it to be true if it has been true at any time during the period and false if it has been false during the entire period.

To facilitate these calculations, we create a class that maintains information about all measured values:

```
public class Record
{
  private DateTime timestamp;
  private double temperatureC;
  private double lightPercent;
  private bool motion;
  private byte rank = 0;

  public Record (DateTime Timestamp, double TemperatureC,
    double LightPercent, bool Motion)
  {
    this.timestamp = Timestamp;
    this.temperatureC = TemperatureC;
    this.lightPercent = LightPercent;
    this.motion = Motion;
  }
}
```

The only field here that requires some comment is the `rank` field. When creating the record, the value that will be set is `0`. For each averaging division, the value of `rank` will be increased by one. So, when calculating a minute average across samples every second, it will be ranked one. When calculating an hour average across minute averages, it will be ranked two, and so on.

We also need to add simple `get` and `set` properties for our fields:

```
public DateTime Timestamp
{
  get { return this.timestamp; }
  set { this.timestamp = value; }
}

public double TemperatureC
{
  get { return this.temperatureC; }
  set { this.temperatureC = value; }
}

public double LightPercent
{
  get { return this.lightPercent; }
  set { this.lightPercent = value; }
}

public bool Motion
{
  get { return this.motion; }
  set { this.motion = value; }
}

public byte Rank
{
  get { return this.rank; }
  set { this.rank = value; }
}
```

Now, let's define the sum of two records this way: if either of the records is `null`, the sum will be the value of the other record. If both contain valid object references, the fields are summed up as follows: the sum of the `timestamp` values is the largest one. Likewise, the sum of the `rank` values is the largest of the two. The sum of the `temperature` and `light` values is the arithmetic sum of each other respectively. The sum of the `motion` property is the logical OR of the two. We formalize this with the following code:

```
public static Record operator + (Record Rec1, Record Rec2)
{
  if (Rec1 == null)
    return Rec2;
  else if (Rec2 == null)
    return Rec1;
  else
  {
    Record Result = new Record (
      Rec1.timestamp > Rec2.timestamp ?
        Rec1.timestamp : Rec2.timestamp,
      Rec1.temperatureC + Rec2.temperatureC,
      Rec1.lightPercent + Rec2.lightPercent,
      Rec1.motion | Rec2.motion);

    Result.rank = Math.Max (Rec1.rank, Rec2.rank);
    return Result;
  }
}
```

We also define a division operator where we divide a record with an integer number to be able to later calculate average values. The `temperature` and `light` values are divided arithmetically, while the `timestamp` and the logical `motion` values are left as they are. The `rank` value is incremented once to give it the property mentioned at the beginning of this section. We formalize this with the following code:

```
public static Record operator / (Record Rec, int N)
{
  Record Result = new Record (Rec.timestamp,
    Rec.temperatureC / N, Rec.lightPercent / N,
  Rec.motion);
  Result.rank = (byte)(Rec.rank + 1);
  return Result;
}
```

Storing historical averages

Before we can calculate historical averages, we need a place to store them. First, we need to add a reference to `System.Collections.Generic` to permit us to use generic list structures:

```
using System.Collections.Generic;
```

We will then add the following static member variables that will be used in our average calculations:

```
private static Record sumSeconds = null;
private static Record sumMinutes = null;
private static Record sumHours = null;
private static Record sumDays = null;
private static int nrSeconds = 0;
private static int nrMinutes = 0;
private static int nrHours = 0;
private static int nrDays = 0;
```

Then, we will add the following static member variables to keep a record of historical averages over time:

```
private static List<Record> perSecond = new List<Record> ();
private static List<Record> perMinute = new List<Record> ();
private static List<Record> perHour = new List<Record> ();
private static List<Record> perDay = new List<Record> ();
private static List<Record> perMonth = new List<Record> ();
```

The idea is to do the following: store each sample in `perSecond` and also sum it up into `sumSeconds`. At the end of each minute, the sum over the second is used to calculate the average of that minute. This average is added to `perMinute` and also summed to `sumMinutes`. At the end of each hour, the sum over the minute is used to calculate an average for the respective hour. This average is added to `perHour` and also summed to `sumHours`, and so on, for hours, days, and months. The code to do this will follow. We start by creating a record containing momentary values. This record will be ranked zero. We add this directly to the sample timer method following the calculation of momentary values:

```
DateTime Now = DateTime.Now;
Record Rec, Rec2;

Rec = new Record (Now, temperatureC,
  lightPercent, motionDetected);
```

We then add this record to historical records, maintaining at most only a thousand records, and sum it to the second-based sum register, as follows:

```
perSecond.Add (Rec);
if (perSecond.Count > 1000)
  perSecond.RemoveAt (0);

sumSeconds += Rec;
nrSeconds++;
```

If it is the start of a new minute, we calculate the minute average and store it in the historical record, maintaining at most a thousand records. We also sum the result to the minute-based sum register and initialize the second-based average calculation for a new minute, as follows:

```
if (Now.Second == 0)
{
  Rec = sumSeconds / nrSeconds;    // Rank 1
  perMinute.Add (Rec);

  if (perMinute.Count > 1000)
  {
    Rec2 = perMinute [0];
    perMinute.RemoveAt (0);
  }

  sumMinutes += Rec;
  nrMinutes++;

  sumSeconds = null;
  nrSeconds = 0;
```

The same is then done again at the start of a new hour. An hour average is calculated and stored, the hour-based sum register is incremented accordingly, and a new period is initialized:

```
  if (Now.Minute == 0)
  {
    Rec = sumMinutes / nrMinutes;
    perHour.Add (Rec);

    if (perHour.Count > 1000)
    {
      Rec2 = perHour [0];
      perHour.RemoveAt (0);
    }

    sumHours += Rec;
    nrHours++;

    sumMinutes = null;
    nrMinutes = 0;
```

The same is done again at the start of a new day. A day average is calculated and stored, the day-based sum register is incremented accordingly, and a new period is initialized:

```
        if (Now.Hour == 0)
        {
          Rec = sumHours / nrHours;
          perDay.Add (Rec);

          if (perDay.Count > 1000)
          {
            Rec2 = perDay [0];
            perDay.RemoveAt (0);
          }

          sumDays += Rec;
          nrDays++;

          sumHours = null;
          nrHours = 0;
```

At the start of a new month, we content ourselves by only calculating the month average and storing it, and initializing a new period. At this point, we don't concern ourselves with removing old values:

```
          if (Now.Day == 1)
          {
            Rec = sumDays / nrDays;
            perMonth.Add (Rec);

            sumDays = null;
            nrDays = 0;
          }
        }
      }
    }
```

C
Object Database

This appendix shows how to persist data in an object database by simply using class definitions. It uses the `Sensor` project example to show how sampled and historical data records are persisted and accessed through the use of an object database proxy.

Setting up an object database

The Raspberry Pi and the Raspbian operating system come with SQLite, a small, flexible SQL database. The `Clayster.Library.Data` library, a powerful and flexible object database, can use this database (and others) to automatically persist, load, and search for objects directly from their class definitions. There is no need to do database development if you are using this library to persist data. To use this object database, we first need to add a reference to the library in our main application with the following code:

```
using Clayster.Library.Data;
```

We then create an internal static `ObjectDatabase` variable that can be used throughout the project, as shown in the next code:

```
internal static ObjectDatabase db;
```

The `db` variable will be our proxy to the object database.

During application initialization, preferably early in the initialization, we tell the object database library what database to use. This can be done either in an application `config` file or directly from the code, as in the following example:

```
DB.BackupConnectionString = "Data Source=sensor.db;Version=3;";
DB.BackupProviderName = "Clayster.Library.Data.Providers." +
  "SQLiteServer.SQLiteServerProvider";
```

The provider's name is simply the full name of the object database provider that will be used, in this case, the object database provider for SQLite. The term "backup" in this case means that this value will be used if no value is found in the application configuration file. We also need to provide a connection string whose format depends on the provider chosen. Since we've chosen SQLite, all we need to do is provide a filename for our database and the version of the library to use. We are then ready to create our object database proxy, as follows:

```
db = DB.GetDatabaseProxy ("TheSensor");
```

The `db` variable is now our proxy. The parameter we send to the `GetDatabaseProxy()` method is the name of the owner of the proxy. Owners can be used to separate data. One owner cannot access data from another owner. The owner can be a simple name or the full name of the class owning the data, and so on.

Database objects

The object database can store almost any object whose class is **Common Language Specification compliant** (**CLS-compliant**). To facilitate the handling of database objects, however, the `Clayster.Library.Data` library provides a base class that can be used and that provides some basic functionality, such as the `SaveNew()`, `Delete()`, `Update()`, and `UpdateIfModifed()` methods, and it provides `OwnerId`, `ObjectId`, `Created`, `Updated`, and `Modified` attributes that can be used to manage or reference objects.

Since it is historical data we want to persist, we will update our `Record` class by making it a descendant of `DBObject`. We also add an attribute to the class, stating that if supported by the object database provider, objects of this class should be persisted in dedicated tables. SQLite does not support dedicated tables, but if you want to change the provider to MySQL, for instance, the provider will support dedicated tables. When creating classes that will be stored in an object database, it is better to do it without preference to what database provider you use at the time of developing the class. The `Record` class is updated using the following code:

```
[DBDedicatedTable]
public class Record : DBObject
```

Each class that is to be used in object databases is required to have a public default constructor defined, otherwise the class cannot be loaded. A default constructor is a constructor without parameters. We define one for our `Record` class as follows:

```
public Record()
  : base(MainClass.db)
{
}
```

Note here that we already limit the class to belong to a particular object database proxy, and therefore a particular owner. If you are sharing the object database with other applications, they cannot access these objects, and vice versa.

We update the existing constructor in a similar way, making sure the owner is set to our object database proxy:

```
public Record (DateTime Timestamp, double TemperatureC,
  double LightPercent, bool Motion)
  : base (MainClass.db)
```

Loading persisted objects

We also add a static method to the `Record` class that allows us to load any objects of this class given a particular parameter, `Rank`, and sort them in the ascending timestamp order, as follows:

```
public static Record[] LoadRecords (Rank Rank)
{
  DBList<Record> List = MainClass.db.FindObjects<Record>
    ("Rank=%0%", (int)Rank);
  List.Sort ("Timestamp");
  return List.ToArray ();
}
```

We also need to define the `Rank` enumeration, remembering our definition of `Rank` earlier. This can be done with the following code:

```
public enum Rank
{
  Second = 0,
  Minute = 1,
  Hour = 2,
  Day = 3,
  Month = 4
}
```

During application initialization, we also need to load any objects persisted earlier. This needs to be done after object database initialization and before the HTTP server is initialized and new samples are made. We will not persist second values in this application, so we start by loading minute values as follows:

```
Log.Information ("Loading Minute Values.");
perMinute.AddRange (Record.LoadRecords (Rank.Minute));
```

We do the same with hourly values:

```
Log.Information ("Loading Hour Values.");
perHour.AddRange (Record.LoadRecords (Rank.Hour));
```

We also load the daily values:

```
Log.Information ("Loading Day Values.");
perDay.AddRange (Record.LoadRecords (Rank.Day));
```

Finally, we do the same with monthly values:

```
Log.Information ("Loading Month Values.");
perMonth.AddRange (Record.LoadRecords (Rank.Month));
```

We also need to initialize our averaging calculations for the different time bases. We begin by initializing our averaging calculations based on minute values. It is only necessary to include records from the same hour as the current hour. Since records are sorted in ascending time order, we simply traverse the list backwards while we remain in the hour as the current one:

```
int Pos = perMinute.Count;
DateTime CurrentTime = DateTime.Now;
DateTime Timestamp;

while (Pos-- > 0)
{
  Record Rec = perMinute [Pos];
  Timestamp = Rec.Timestamp;
  if (Timestamp.Hour == CurrentTime.Hour && Timestamp.Date ==
    CurrentTime.Date)
  {
    sumMinutes += Rec;
    nrMinutes++;
  }
}
else
  break;
```

We do the same operation with hourly and daily values as well, without showing it explicitly here.

Saving and deleting objects

The only thing missing now is to save new `Record` objects that we create and then delete old objects we no longer want to keep. To save a new object, we simply call the `SaveNew()` method on the object, as follows:

```
perMinute.Add (Rec);
Rec.SaveNew ();
```

Note that we repeat the previous code also for new hourly, daily, and monthly values. And to delete an old object, we only call the `Delete()` method, as follows:

```
perMinute.RemoveAt (0);
Rec2.Delete ();
```

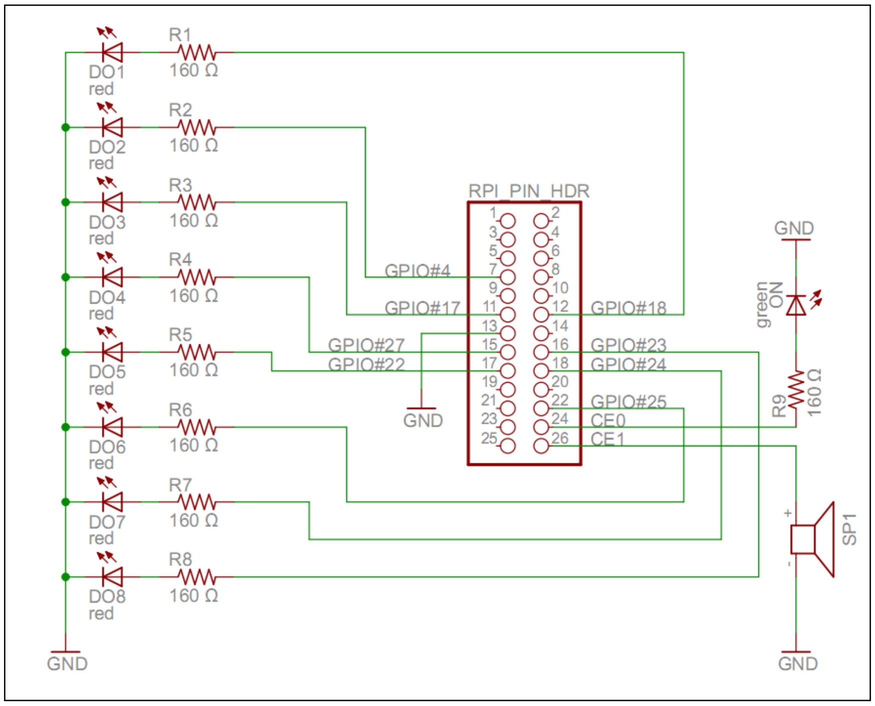
Note that a new object can only be saved using `SaveNew()` once. Afterwards, the `Update()` or `UpdateIfModified()` methods have to be used, if updates are made to the object.

We can now run our application again, let it run for a while, reset the Raspberry Pi, rerun the application, and see that the previous values are still available.

D
Control

Performing basic control operations is a crucial task for any actuator. This appendix shows you how control operations are implemented in the actuator project published in the book. You start by creating a project as described in *Appendix A*, *Console Applications*, and then follow it up with the instructions in this appendix.

Here, we will start by interfacing our hardware, configuring it, preparing the code with the basic data structures, and then starting sampling values sensed by the hardware. The circuit diagram for our prototype board, as described in *Chapter 1*, *Preparing our IoT Projects*, is as follows:



Interfacing the hardware

All hardware, except the alarm output, are simple digital outputs. These can be controlled by the `DigitalOutput` class. The alarm output will control the speaker through a square wave signal that will output on the **GPIO#7** pin, using the `SoftwarePwm` class, which outputs a **pulse width modulated** (**PWM**) square signal on one or more digital outputs. The `SoftwarePwm` class will only be created when the output is active. When not active, the pin will be left as a digital input.

The declarations look as follows:

```
private static DigitalOutput executionLed =
  new DigitalOutput (8, true);
private static SoftwarePwm alarmOutput = null;
private static Thread alarmThread = null;
private static DigitalOutput[] digitalOutputs =
  new DigitalOutput[]
{
  new DigitalOutput (18, false),
  new DigitalOutput (4, false),
  new DigitalOutput (17, false),
  new DigitalOutput (27, false),
    // pin 21 on Raspberry Pi R1
  new DigitalOutput (22, false),
  new DigitalOutput (25, false),
  new DigitalOutput (24, false),
  new DigitalOutput (23, false)
};
```

Controlling the alarm

The alarm will be controlled from a separate low-priority thread. We make sure it is below normal priority so that it does not affect network communication and other more important tasks. To turn the alarm on, we call the following method:

```
private static void AlarmOn ()
{
  lock (executionLed)
  {
    if (alarmThread == null)
    {
      alarmThread = new Thread (AlarmThread);
      alarmThread.Priority = ThreadPriority.BelowNormal;
      alarmThread.Name = "Alarm";
```

```
        alarmThread.Start ();
      }
    }
  }
```

To turn it off, we call this method:

```
  private static void AlarmOff ()
  {
    lock (executionLed)
    {
      if (alarmThread != null)
      {
        alarmThread.Abort ();
        alarmThread = null;
      }
    }
  }
```

The thread controlling the alarm will only create the PWM output on the GPIO pin 7 and then oscillate the output frequency to generate an alarm sound. The duty cycle will be maintained at 0.5, meaning that 50 percent of the time the wave will be high, and during the remaining 50 percent it will be low. The oscillation starts at 100 Hz, is increased to 1,000 Hz in steps of 10 Hz per 2 milliseconds, and then lowered back to 100 Hz, and so the process is repeated:

```
  private static void AlarmThread ()
  {
    alarmOutput = new SoftwarePwm (7, 100, 0.5);
    try
    {
      while (executionLed.Value)
      {
        for (int freq = 100; freq < 1000; freq += 10)
        {
          alarmOutput.Frequency = freq;
          System.Threading.Thread.Sleep (2);
        }
        for (int freq = 1000; freq > 100; freq -= 10)
        {
          alarmOutput.Frequency = freq;
          System.Threading.Thread.Sleep (2);
        }
      }
    }
```

```
      catch (ThreadAbortException)
      {
        Thread.ResetAbort ();
      }
      catch (Exception ex)
      {
        Log.Exception (ex);
      }
      finally
      {
        alarmOutput.Dispose ();
      }
    }
```

Since the alarm is turned off by calling the `AlarmOff()` method, which aborts the execution of the thread, we make sure to catch the `ThreadAbortException` exception to make a graceful shutdown of the thread.

Features adapted from the sensor project

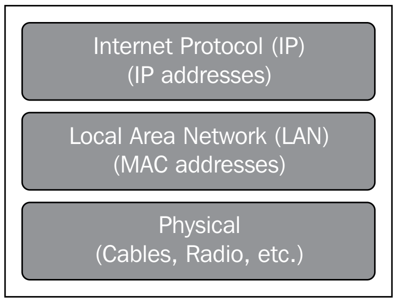
The following topics will not be discussed explicitly for the actuator project since they are implemented in a similar manner as for the sensor project:

- Main application structure
- Event logging
- Export of current output states as sensor data
- User credentials and authentication
- Connection to an object database
- Persistence of output states
- Deployment and execution after the system is restarted

E
Fundamentals of HTTP

As long as communication protocols are concerned, the success of the **Hypertext Transfer Protocol** (**HTTP**) is only eclipsed by the pervasive success of the **Internet Protocol** (**IP**), the fundamental communication protocol on the Internet. While IP was developed almost two decades earlier, and is used by all protocols communicating on the Internet, it lives a relatively anonymous life compared to HTTP among the broader public.

IP is basically used to route packets between machines (or hosts) on the Internet, knowing only the IP address of each machine. Traditionally, networks were local, and each of the connected machines could only communicate with other machines on that same network using a specific address depending on the type of network used. Today, such networks are known as **Local Area Networks** (**LAN**), and the most commonly used LAN networks are of type *Ethernet*, which uses **Media Access Control** (**MAC**) addresses as local network addresses. Using the IP protocol and IP addresses, it was possible to inter-connect different networks, and make machines communicate with each other regardless of to what type of local area network they were connected. And thus the **Internet** was born; communication could be made between machines on different networks. The following diagram shows the relationship between IP, LAN, and the Physical network, in what is called a protocol stack diagram:

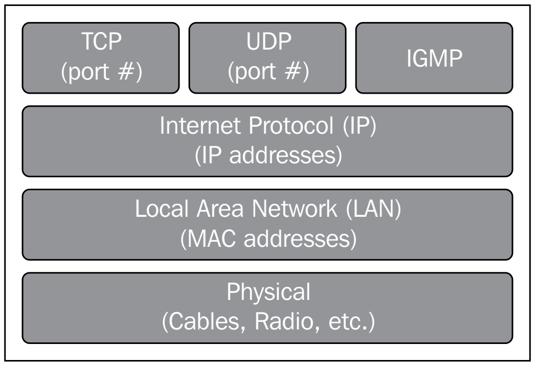


Communicating over the Internet

IP is often mentioned together with the **Transmission Control Protocol** (**TCP**), in the form of TCP/IP. TCP allows the creation of **connections** between machines. Each connection endpoint is identified by the IP address of the machine and a *Port number*. Port numbers allow thousands of connections to be made to or from a single machine. There are both well-known standardized port numbers that are used by well-known services as well as private port numbers that are short-lived port numbers for private use. Packets sent over a TCP connection are furthermore guaranteed to be delivered in the same order as they were sent, and without packet loss, as long as the connection is alive. This makes it possible to create data streams, where large streams of data can be sent between machines, in a simple manner and without regard to details such as packet size, retransmissions, and so on.

TCP has an important cousin, the **User Datagram Protocol** (**UDP**). UDP also transmits packets (called **datagrams**) between machines using IP and port numbers, but without using connections and retries to assure datagram order and delivery. This makes UDP much quicker than TCP, and is often preferred in favor of TCP in cases where a certain degree of packet loss is not a problem, or handled explicitly by the overlying service or application.

UDP is also often used together with the **Internet Group Management Protocol** (**IGMP**) to allow the transmission of datagrams to multiple recipients at once, without having to send the datagram to each recipient individually. This manner of transmitting packets is called **multicasting**, as opposed to **unicasting** where packets are sent from one machine to another. Using multicasting, it is sufficient to transmit a stream once on a backbone, regardless of how many recipients are connected to the backbone. If IGMP-enabled routers or switches are used when connecting to the backbone, each local area network is not congested with all streams transmitted on the backbone, only the streams actively subscribed to, on the local area network. The following diagram shows the relationship between the protocols in a protocol stack diagram:



From an application point of view, the operating system provides it with network sockets, where the application can choose what protocol to use (typically TCP or UDP), which machine to communicate with (IP address), and what port number to use. To make life easier for end users, **Domain Name Servers** (**DNS**) are used to provide hosts in the IP network with a name that applications can refer to. The operating system normally provides the application with the possibility to use host names instead of IP addresses in all application programming interfaces.

The creation of HTTP

Originally developed as a means to transport (scientific) text documents containing simple formatting and references to other documents, the HTTP protocol is used in a much broader context today. These documents were written in **Hypertext Markup Language** (**HTML**) and the protocol was thus called HTTP. At that time, the mark-up language contained simple formatting, but could also include references to images seen inside the document and references to other documents (so called *links*).

Locating resources on the World Wide Web

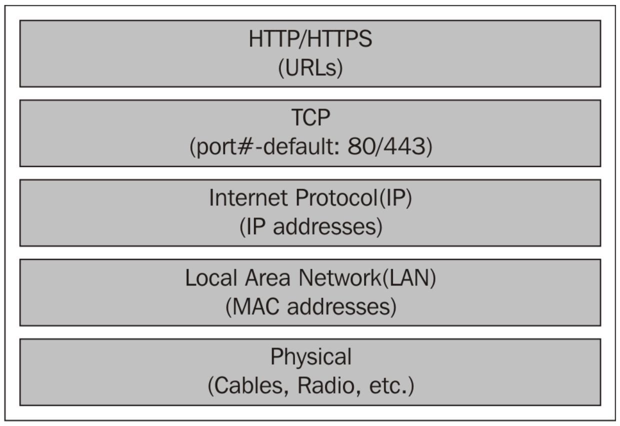
When HTTP was first invented, content on the Web was seen as files made accessible by hosts. Each content item would be assigned a **Uniform Resource Locator** (**URN**), so that links or references could be made to each content item. Over time, web resources have evolved a great deal to include dynamically generated content based on queries, and so on.

Uniform Resource Locators for use with HTTP(S) are made up of five parts: first, a **Uniform Resource Identifier scheme** (**URI scheme**), HTTP or HTTPS. This URI scheme identifies which versions of the protocol to use. (URIs and URLs are often confused and intermixed. URIs are used for identification of resources, and do not necessarily point to a location where the resource can be found. A URL is a location where the resource can be accessed.) Following the URI scheme, comes the authority, which in this case is either the domain name or IP address of the host publishing the content item, optionally followed by the port number, if not the standard port number for the protocol. The third part is the path of the content item, similar to a local file path. Following the path are optional query parameters and an optional fragment identifier.

The following figure provides an example of the different parts of an URL/URI:

$$\underbrace{http}_{scheme} \; :// \underbrace{\overbrace{example.com}^{domain} : \overbrace{8080}^{port}}_{authority} \; \underbrace{/d/1.htm}_{path} \; ? \; \underbrace{q1=v1}_{query} \; \# \; \underbrace{f0011}_{fragment}$$

HTTP uses TCP to communicate. Communication is either performed over an unencrypted connection (in which the URI scheme is HTTP). In this case, the default port number is 80. Or communication can be performed over an encrypted channel (in which the URI scheme is HTTPS). In this case, the default port number is 443. The following diagram shows HTTP and HTTPS in a protocol stack diagram:



Securing communication using encryption

When using HTTPS, encryption is performed using **Secure Sockets Layer** (**SSL**), which evolved into **Transport Layer Security** (**TLS**). These in turn use X.509 Certificates to handle identification and actual encryption. These certificates provide three services.

Firstly, they use a **Public Key Infrastructure** (**PKI**) encryption algorithm that provides two keys, one public and one private. The *public key* can be sent to whoever wants to communicate with the owner of the certificate. With this public key, the encryption algorithm used by the certificate can encrypt information. But you need the *private key* to be able to decrypt it.

Secondly, each certificate contains information that can be used to identify the holder (or **subject**) of the certificate. For web servers, this is typically the domain name of the server. When you connect to a web server and it returns a certificate, you can use that certificate to make sure you're talking to the correct web server and nothing else.

Thirdly, all certificates contain information of who its creator (or issuer, or certificate authority) was. Following these links, from certificate to its issuer, and so on, until you reach a root certificate, you get a chain of trust. When validating a certificate, this chain of trust is processed. Each issuer has the possibility to inform the entity performing certificate validation, that the corresponding certificate has been revoked. If the private key of a certificate has been compromised, the certificate must be revoked, and a new certificate created instead. By revoking a certificate with its issuer, you make sure nobody can use the certificate illicitly as long as everybody makes sure to validate all certificates used.

Normally, if HTTPS is used, only the server provides a certificate to identify itself to the client. If the client chooses to use HTTP to secure the communication, it should properly validate the server certificate to make sure the server is who it is, and not a **man-in-the-middle** (**MITM**); somebody pretending to be the server, to eavesdrop on the conversation. The client can also provide a client-side certificate to authenticate itself to the server. But since certificates are complicated to create, require maintenance, and often incur a cost and require a high level of knowledge of the user operating the client, other methods are often used to authenticate the client. Such methods are discussed later in this chapter. Certificates are most often used only by high-value entities, such as servers.

You can create self-signed certificates, which are basically certificates without an issuer. These can only be used if certificate validation is not performed or if the certificate is installed as a root certificate by each party validating it. This should be avoided, since this may create security issues elsewhere in the system, especially if the certificate store used is shared between applications.

Requests and responses

HTTP is based on the request/response communication pattern, where a client makes a request to a server and the server responds to the request by sending a response. Each request starts by stating what method to use followed by the resource (path and query) followed by the protocol version used (1.0, 1.1 or 2.0). Afterwards follows a sequence of text headers followed by an optional data section. The headers contain information about how optional data is encoded, what data is expected in the result, user authentication information, cookies, and so on.

Depending on the method used in the request, the server is expected to act differently. The most common methods used are: The `GET` method fetches data from the server. The `HEAD` method tests whether data is available on the server by simulating a `GET` method but only returning the headers. The `POST` method posts data to the server, for example data in a form. The `PUT` method uploads content, for example uploads a file. The `DELETE` method removes content, for example deletes a file. The `OPTIONS` method can be used to check to see what methods are supported by the server or a resource.

The server responds to the request in a similar way; first by returning the protocol version supported by the server, followed by a status code and a status message. After the status code and message, follows a sequence of text headers, followed by an optional data section. In this case, the headers include not only how to encode the data, but for how long it is valid, and so on. Other headers control authentication, cache control and cookies, and so on.

While HTTP 1.0 and 1.1 (which are the versions mainly supported at the time of writing this book) only support one request/response operation at a time over a single connection, and only from the client to the server, future versions of the protocol (such as version 2.0 currently being developed) will support multiple simultaneous operations over a single connection. It will also support bidirectional communication.

Status codes

There are several status codes defined for use in HTTP. Are they important to remember? Some are very well known, such as the 404 Not Found, which has turned into its own meme, while others a bit more obscure.

It is often sufficient to know that 1xx status codes are informational, 2xx status codes imply success (of some kind), 3xx status codes imply a redirection (of some kind), 4xx status codes imply a client-side error in the request on behalf of the client, and 5xx status codes imply server errors that occur on the server. Some of the more important codes are listed in the following table. However, since they are necessary to correctly implement server-side web resources you should not feel limited by this list; a complete list can be found at `http://tools.ietf.org/html/rfc2616#section-6.1.1`.

Code	Message	Meaning
`200`	`OK`	Operation successful.
`301`	`Moved Permanently`	Resource moved permanently. Update original URL.

Code	Message	Meaning
`303`	`See Other`	Used in the PRG pattern (POST/Redirect/GET), to avoid problems in browsers. There's more about this later on.
`307`	`Temporary Redirect`	Redirect to another URL. No need to update original URL.
`308`	`Permanent Redirect`	Redirect to another URL. Update original URL.
`400`	`Bad Request`	The request made was badly formed.
`401`	`Unauthorized`	User has not been authenticated and cannot reach a resource that requires authentication.
`403`	`Forbidden`	User has been authenticated but lacks privileges to access resource.
`404`	`Not Found`	Resource not found on server.
`500`	`Internal Server Error`	An exception occurred on the server during the processing of the request.

Encoding and transmitting data

Clients and servers tell each other how to interpret the data that optionally follows the headers using a set of header key and value pairs. The `Transfer-Encoding` header can be used to tell the recipient the size of the content is not known at the time of sending the headers, and is therefore sent in **chunks**. Chunked communication allows for dynamically generated content, where content is sent as it is being generated. `Content-Encoding` can be used to send compressed data. `Content-Type` describes how the actual content is encoded and decoded as a binary stream of bytes. If the number of bytes used in transmitting the content is not implicitly defined by the context, such as when using chunked transfer, the number of bytes of the encoded content must be sent to the recipient using the `Content-Length` header.

`Content-Type` is closely related to **MIME** (short for **Multipurpose Internet Mail Extensions**) types, originally developed for encoding content in mail. Today, it is common to discuss **Internet Media Types** (IMT), instead of specifically discussing MIME types (for mail) or content types (for the Web).

IMT, in its common form consists of a type and a subtype, in the form `type/subtype`. Sometimes, the subtype is classified more, using a suffix, as follows: `type/subtype+suffix`. Common types include text, image, audio, video, application, and so on. One that is not self-explanatory is the application type, which does not necessarily contain applications, but application data for applications. The following table shows some common media types to illustrate the concept.

IANA maintains a full list of registered media types that can be found at `http://www.iana.org/assignments/media-types/media-types.xhtml`.

Media type	**Description**
`application/atom+xml`	ATOM media feeds.
`application/json`	JSON-formatted data.
`application/rdf+xml`	RDF-formatted data.
`application/soap+xml`	SOAP-formatted data (web service calls and responses).
`application/x-www-form-urlencoded`	Used to encode form data containing only simple form parameters.
`audio/mpeg`	MP3 or other MPEG audio.
`image/jpeg`	JPEG-encoded image.
`multipart/form-data`	Used to encode form data including files, images, and so on, posted to a server.
`text/plain`	Plaint text file.
`text/html`	HTML file.
`text/xml`	XML file.
`video/mp4`	Mpeg-4 encoded video file.

States and sessions in HTTP

HTTP is considered to be **stateless** by itself, which means that the server responding to requests does not by itself remember anything from previous conversations with the client. This also means that the request must contain all information the server requires to process the request. Stateless protocols simplify scaling, in that you can have multiple machines serving requests to the same domain. But often it is not a good idea to either let the client maintain all information and resend it in each request to the server. Consider, for instance, browsing through a large result set from a database search operation. Should the server search the database again when navigating in the result set? Or should the client contain the entire result set (which might take time to download), even though they may be many more than the user is interested in? To solve such issues, applications running on web servers (as HTTP servers are also called) on top of the HTTP layer create the concept of a session. A session is a server-side construct, where the application can store information. The session is identified using an identifier (which can be application-specific), and sends only the session identifier to the client in the form of a cookie. (Cookies can contain any type of information, not only session identifiers.) To work correctly, clients need to remember what cookies it has received, from where they received them, and for how long they are valid.

When making new requests to a server, it provides any cookies it has received pertaining to that particular server, the server forwards the cookies to the application, and the application can continue processing the request using any state information available in the implied session. Another important use of sessions and cookies is to maintain user credentials. When a user logs into a service, information about the user's credentials and privileges will be stored in the session. This allows the user to navigate a site and the servers involved will be able to adapt the contents to the user, and their privileges and personal settings.

User authentication

User authentication is important in secure applications and requires the user (human or machine) behind the client connecting to a server to authenticate its credentials to the server. In theory, client-side certificates over an encrypted connection could be used to achieve this. This is also done when high-value entities communicate between each other. But in web applications or IoT applications where masses of end users without technical skills or low-value entities, such as small things, want to communicate, certificates do not offer a practical solution.

HTTP has a built-in authentication mechanism called **WWW-authentication** to differentiate itself from the more commonly known **Simple Authentication and Security Layer** (**SASL**) used in other Internet protocols. Even though it is technically different, it works in similar ways as SASL, by allowing multiple and pluggable authentication mechanisms to be used, and by allowing the client to decide which method to use, restricted only to the list of methods provided by the server.

Even though such a method works well in automation, where it is easy for clients to authenticate themselves repetitively, it has several drawbacks when it comes to human users.

The first drawback is that it is implemented on a protocol level, while sessions are implemented on the application level since the protocol is stateless. This means that the web server (or web client) are not aware that a session exists and that a user is logged in or already authenticated by the application. This implies that unless special provisions are taken to bypass this logic, the user needs to authenticate themselves repetitively every time a new resource is fetched. As mentioned previously, this is not necessarily a problem for machines. (It can even be an advantage sometimes.) To avoid such repetitive user authentications, browsers attempt to store user credentials so that the browser can respond by itself, without obviating the end user. But this is in itself a security issue, since it bypasses the requirement that the real end user responds to the login challenge, and not the browser. How does the server know the difference between the true user opening up a browser with a stored password, and another person using the same browser to look at the same page?

The second drawback of using WWW-authentication for human users is its lack of customization of user interfaces. Again, this is not a problem for machines. But the previously mentioned reasons are more than sufficient for web developers to want to implement user authentication by themselves, in the application layer. It also ties into session management in a more logical manner.

Web services and Web 2.0

As the World Wide Web was formed, and HTTP became popular, it became obvious that it was difficult and costly to maintain and publish interesting content on a static web. The model that users used clients to browse existing information was not sufficient. There was a need to let users be able to interact more directly with available applications running on web servers. There was also a need to automate content publication on servers, which meant to be able to go beyond the quite limited possibilities that existed at the time. This included content provided by online web forms, content published by uploading files to the web server, or out-of-band (non HTTP-based) methods. There was a need to communicate with underlying web applications in a more efficient manner.

With the development of XML, the web community had an exceptional tool to encode any type of data in a structural manner. XML schemas can be used to validate XML to make sure it is formatted as it should be. Since XML has a well-known media type, web clients and web servers know how to encode and decode it, and it could thus be easily published by web servers, downloaded, and used to customize user experiences. It can be automatically transformed using XSL Transformations (XSLT) into anything based on text (such as HTML), and supports all kinds of features. But it can also be uploaded (using POST) to web applications to send data to it, without updating actual application files. And for this automation got a great tool to automatically call services within an application, and thus web services and **Service-Oriented Architecture** (**SOA**) was born. Little was it known at the time (which seems to be a tradition for the Web) that this would dramatically change the World Wide Web, and how users interact with applications. Today, the advent of web services can be seen as the birth of Web 2.0, and is the basis for everything from applications only hosting end-user generated content to most smart applications running on smartphones. Today, web applications are not necessarily HTML and script-based applications running in browsers, but can be native smartphone applications communicating with their web servers using web services. This is also the basis for automation and Internet of Things, over HTTP.

SOAP or REST

There are several different types of web services available and two are well known and commonly used today: the first one is called **Simple Object Access Protocol** (**SOAP**) and the second is called **Representational State Transfer** (**REST**).

Once all XML-based technologies were developed, it was a relatively simple task to start to standardize how web service calls should be made and how responses should be returned. It resulted not only in schemas for how the actual calls and responses are made (these are called SOAP), but also in schemas for how to document these calls (these are called **Web Service Definition Language** (**WSDL**)). With WSDL, it was not only possible to automate the actual calls themselves but also to automate the actual coding or implementation of the calls. Most developer tools today allow you to create a web reference, which basically downloads the WSDL document from the web server and the tool automatically generates code that will make the corresponding calls described in the document.

Automation never looked simpler. Or, at least, until the first update to the model had to be done. One of the major problems with SOAP-based web services is that it creates a *hardly coupled link* between the client and the server. If you update one link, it is likely to break the other, unless special care is taken. If you do not have the control of all participants using the web service, versioning and compatibility issues become a major problem. This is especially true for web applications, which grow and change dynamically from their inception until they mature.

The development of RESTful web services was a reaction to the rigor of SOAP, which is shown in the acronym itself. Instead of attempting to solve all problems in one protocol on top of HTTP, the idea was to go back to the roots of HTTP and allow developers to use simple HTTP actions to create web service calls to the underlying application. This could include simply encoding the call in the URL itself, or by posting a simple (proprietary) XML document to a specific URL, encoding which method to call. RESTful web services also allow methods to dynamically generate content in a freer sense than what is allowed in SOAP. Furthermore, RESTful web services do not have the same problems with versioning since it is easier to aggregate parameters and features without breaking existing code. And last not least, it is often possible to call RESTful web services from a browser, without the need for special tools. On the Web, it has been shown that *loosely coupled* interfaces (such as RESTful interfaces are) win over hardly coupled interfaces, even if the hardly coupled interfaces provide more functionality, at least when it comes to web services.

The Semantic Web and Web 3.0

Before we finish the theoretical overview of the HTTP protocol and start looking how to practically use it in applications for Internet of Things, it is worth mentioning recent (and some not so recent) developments in the field.

As more and more communication was done over HTTP that was not related to fetching hypertext documents over the Internet; it was understood that the basic premise and original abstraction of the Web was needed. Instead of URLs pointing to "pages", which is a human concept, URLs should point to data, or even better all types of data should be able to be identified using URIs, and if possible, even URLs. This change in abstraction is what is referred to as linked data.

But what is the best way to represent data? At that time, data could be encoded using XML, but this didn't mean it could be "understood" or processed, or meaningful relationships extracted between distributed sets of data. A different method was needed. It was understood, that all knowledge humans can communicate in language, can be expressed, albeit not in a Nobel peace prize winning manner, using triples consisting of a subject (who does or relates something), a predicate (what is happening, relating, or being done) and an object (on or to what is something being done or related). All three are represented by URIs (or URLs), and objects can also be literals. If an object is represented by an URI (or URL) it can in turn also be a subject that relates to a large set of objects. The abstraction of all types of data into such Semantic Triples has resulted in the coining of the web of linked data as the Semantic Web.

Data in the Semantic Web is represented either using **Resource Description Framework** (**RDF**), readable by machines, or **Terse RDF Triple Language** (**TURTLE**). As the data representation is standardized, it is possible to fetch and process distributed data in a standardized manner using the **SPARQL Protocol and RDF Query Language** (**SPARQL** for short, pronounced "sparkle"). SPARQL is for distributed data on the web, what SQL is for distributed data in tables in a relational database. In a single operation, you can select, join, and process data from the Internet as a unit, without having to code programs that explicitly fetch data from different locations, join them, and process them before returning the data to the requester.

F
Sensor Data Query Parameters

Sensor data is formed by the following components:

- Each device reports data from one or more nodes. Each node is identified by its node identifier. In larger systems, nodes might be partitioned into data sources. In this case, nodes are identified by a source identifier and a node identifier. In even larger systems, nodes are identified using a triple of source identifier, cache type, and node identifier. But for all our purposes, it is sufficient to identify nodes by their node identifiers.
- Each node-reporting sensor data does so with timestamps. No data can be reported without a valid timestamp.
- For each timestamp representing a point in time, one or more sensor data fields are reported. These fields can be numerical, string-valued, Boolean-valued, date- and time-valued, timespan-valued, and enumeration-valued.
- Each field has a field name. This field name is a string and should be human readable, but at the same time, well defined so that it can be machine understandable. It must not be localized.
- Each field has a value, depending on the type of field it is. Numerical fields also have an optional unit and information about the number of decimals used. In the context of sensor data, 1.210 m^3 is not the same as 1.2 m^3. The first has more precision, the second less. You could not say whether the physical magnitude measured by the second is larger or smaller than the first value, for instance, even though it would probably be larger and the numerical number smaller.

- Each field has a readout type classification, which categorizes as a momentary value, peak value, status value, identification value, computed value, or historical value. If not explicitly specified, it is assumed to be a momentary value.
- Each field has a field status or quality of service level. This specifies whether the value is missing, automatically estimated, manually estimated, manually read, automatically read, offset in time, occurred during a power failure, has a warning condition, has an error condition, is signed, has been used in billing, its bill has been confirmed, or is the last value in a series. If not specified, it is simply assumed the field is an automatically read value.
- Each field has optional localization information, which can be used to translate the field name into different languages.

Often, as in our case, a sensor or meter has a lot of data. It is definitely not desirable to return all data to everybody requesting information. In our case, the sensor can store up to 5,000 records of historical information. How can we motivate exporting all this information to somebody only wanting to see momentary values? We can't. The `ReadoutRequest` class in the `Clayster.Library.IoT.SensorData` namespace helps us parse the sensor data request query in an interoperable fashion and lets the application know what type of data is requested. The following information can be sent to the web resource, and is parsed by the `ReadoutRequest` object:

- Any limitations of what field names to report. In our case, if the requester is only interested in temperature, why send light and motion values as well?
- Any limitations of what nodes to report. In our case, this parameter is not very important since we will only report values using one node, the sensor itself. But in a multinode thing, this parameter tells the thing from which things data should be exported.
- Any limitations on what readout types to report. In our case, is the requester interested in momentary values or historical values, and of which time base?
- Any limitations on what time interval is desired. If only a specific time interval is of interest, it can drastically reduce data size, if the thing has a lot of historical data.
- Any information about external credentials used in distributed transactions. External credentials in distributed transactions will be covered more in detail in later chapters. Sometimes, when assessing who has the right to see what, it is important to know who the final recipient of the data is.

The following table lists the query parameters understood by the `ReadoutRequest` class. Query parameters in this case are case insensitive, meaning that it is possible to mix uppercase and lowercase characters in the parameter names and the `ReadoutRequest` object will still recognize them.

Parameter	**Description**
`nodeId`	This is the ID of a node to read.
`cacheType`	This is the cache type used to identify the node.
`sourceId`	This is the source ID used to identify the node.
`from`	This only reports data from this point in time, and newer data.
`to`	This only reports data up to this point in time, and older data.
`when`	This is used when readout is desired. It is not supported when running as an HTTP server.
`serviceToken`	This is the token that identifies the service making the request.
`deviceToken`	This is the token that identifies the device making the request.
`userToken`	This is the token that identifies the user making the request.
`all`	This is the Boolean value that indicates whether all readout types are desired. If no readout types are specified, it is assumed all are desired.
`historical`	This is the Boolean value that indicates whether all historical readout types are desired, regardless of time base.
`momentary`	This is the Boolean value that indicates whether momentary values are desired.
`peak`	This is the Boolean value that indicates whether peak values are desired.
`status`	This is the Boolean value that indicates whether status values are desired.
`computed`	This is the Boolean value that indicates whether computed values are desired.
`identity`	This is the Boolean value that indicates whether identity values are desired.
`historicalSecond`	This is the Boolean value that indicates whether historical second values are desired.
`historicalMinute`	This is the Boolean value that indicates whether historical minute values are desired.
`historicalHour`	This is the Boolean value that indicates whether historical hour values are desired.

Parameter	**Description**
`historicalDay`	This is the Boolean value that indicates whether historical day values are desired.
`historicalWeek`	This is the Boolean value that indicates whether historical week values are desired.
`historicalMonth`	This is the Boolean value that indicates whether historical month values are desired.
`historicalQuarter`	This is the Boolean value that indicates whether historical quarter values are desired.
`historicalYear`	This is the Boolean value that indicates whether historical year values are desired.
`historicalOther`	This is the Boolean value that indicates whether historical values of another time base are desired.

G
Security in HTTP

Publishing things on the Internet is risky. Anybody with access to the thing might also try to use it with malicious intent. For this reason, it is important to protect all public interfaces with some form of user authentication mechanism, to make sure only approved users with correct privileges are given access to the device.

As discussed in the introduction to HTTP, there are several types of user authentication mechanisms to choose from. High-value entities are best protected using both server-side and client-side certificates over an encrypted connection (HTTPS). But this book concerns itself with things not necessarily of high individual value. But still, some form of protection is necessary.

We are left with two types of authentication; both will be explained in this chapter. The first is the WWW-authentication mechanism provided by the HTTP protocol itself. This mechanism is suitable for automation. The second is a login process embedded into the web application itself, and using sessions to maintain user login credentials. This appendix builds on the Sensor project, and shows how important HTTP-based interfaces are protected using both WWW-authentication for **machine-to-machine** (**M2M**) communication and a login/session based solution for human-to-machine scenarios.

WWW-authentication

To add WWW-authentication to some of our web resources, we begin by adding the following reference at the top of our main application file:

```
using Clayster.Library.Internet.HTTP.ServerSideAuthentication;
```

There are several different types of authentication mechanisms you can choose from. Basic authentication is the simplest form of authentication. Here, the username and password are sent in clear text to the server, which validates them. This method is not recommendable, for obvious reasons. Another mechanism is the digest authentication method. It is considered obsolete because it is based on MD5 hashes. Since a weakness has been found in MD5 (without nonce values), the method is no longer recommended. But it is a simple method, and the MD5 Digest method, using nonce values, still provides some form of security so we will use it here for illustrative purposes. To activate the digest authentication method, register it with the HTTP server as follows:

```
HttpServer.RegisterAuthenticationMethod (
  new DigestAuthentication ("The Sensor Realm",
    GetDigestUserPasswordHash));
```

Registration should be done for the HTTPS server as well. If stronger protection is desired, such methods can be implemented by simply creating a class that inherits from the `HttpServerAuthenticationMethod` base class.

Now that we have registered at least one WWW-authentication method on the server, we flag which web resources must be authenticated this way, before access is granted to the resource. The following code enables WWW-authentication for our sensor data export resources, by sending `true` in the third parameter during registration:

```
HttpServer.Register ("/xml", HttpGetXml, true);
HttpServer.Register ("/json", HttpGetJson, true);
HttpServer.Register ("/turtle", HttpGetTurtle, true);
HttpServer.Register ("/rdf", HttpGetRdf, true);
```

User credentials

Before we can calculate our Digest User Password Hash, needed for the Digest authentication method, we need to know what user credentials are valid first. We will build a very simple authentication model, with only one user. To be able to change the password, we will need a class to persist the credentials. We therefore create a new class to be used with with the object database:

```
public class LoginCredentials : DBObject
{
  private string userName = string.Empty;
  private string passwordHash = string.Empty;

  public LoginCredentials ()
```

```
      : base (MainClass.db)
    {
    }
  }
```

We publish the `UserName` property as follows:

```
[DBShortString(DB.ShortStringClipLength)]
public string UserName
{
  get { return this.userName; }
  set
  {
    if (this.userName != value)
    {
      this.userName = value;
      this.Modified = true;
    }
  }
}
```

Here we note two things: firstly, we place an attribute on the string property, saying it is a **short string**. This means it has a maximum length of 250 characters, which means it can be stored in a certain way, as well as be indexed. **Long strings** can be of any length, but they are stored differently. Secondly, we note the special set method implementation, where we set the **Modified** attribute only if the value changes. By implementing properties in this way, we can use `UpdateIfModified()` instead of `Update()`, and save database access when objects have not changed.

The vigilant observer has already noted that the `UserName` class does not contain a password property, but a password **hash** property. This is very important. Passwords should never be stored anywhere, if you can avoid it. Most Internet authentication mechanisms today support the use of intermediate hash values to be used instead of passwords directly. This allows these hash values to be stored instead of storing the original password. We will take advantage of this fact and only store the password hash. We do this by publishing the property in the following manner:

```
[DBEncryptedShortString]
public string PasswordHash
{
  get { return this.passwordHash; }
  set
  {
    if (this.passwordHash != value)
```

```
            {
              this.passwordHash = value;
              this.Modified = true;
            }
          }
        }
```

Note here that we also add an attribute, letting the object database know that the property is not only a short string, but that it should also be encrypted before storing the value. It can be said that this encryption is considered a weak form of encryption. But at least the data is not stored in clear text.

Loading user credentials

At the end of the class, we add a static method that allows us to load any credential object persisted. Since we only one such object is allowed, we choose to delete any other objects found, created after the first object:

```
public static LoginCredentials LoadCredentials ()
{
  return MainClass.db.FindObjects<LoginCredentials> ().
    GetEarliestCreatedDeleteOthers ();
}
```

In the main class, we create a private static variable that will hold the user credentials object:

```
private static LoginCredentials credentials;
```

During the initialization phase, we also load the object from the object database. If no such object is found, we create a default object, with default user name `Admin`, and a default password `Password`:

```
credentials = LoginCredentials.LoadCredentials ();
if (credentials == null)
{
  credentials = new LoginCredentials ();
  credentials.UserName = "Admin";
  credentials.PasswordHash = CalcHash ("Admin", "Password");
  credentials.SaveNew ();
}
```

Calculating the hash

We are now ready to calculate the hash value to use in our authentication scheme. The digest authentication method stipulates that the password hash must be calculated over the string formed by concatenating the user name with a colon (":") the realm, another colon, and finally the password. We do this, as follows:

```
private static string CalcHash (string UserName, string Password)
{
  return Clayster.Library.Math.ExpressionNodes.Functions.
    Security.MD5.CalcHash (string.Format (
      "{0}:The Sensor Realm:{1}", UserName, Password));
}
```

We are now ready to create the `GetDigestUserPasswordHash()` interface method that the digest method needs, in order to know whether an authenticating user is valid or not:

```
private static void GetDigestUserPasswordHash (string UserName,
  out string PasswordHash, out object AuthorizationObject)
{
  lock (credentials)
  {
    if (UserName == credentials.UserName)
    {
      PasswordHash = credentials.PasswordHash;
      AuthorizationObject = UserName;
    }
    else
    {
      PasswordHash = null;
      AuthorizationObject = null;
    }
  }
}
```

Two comments can be made on the implementation in the preceding code: firstly, the method looks up a username and returns null values if a user with that name is not found. Otherwise, the corresponding password hash is also returned. The digest method only authorizes a user if the password hash provided by the user is equal to the password hash provided by the interface method, `GetDigestUserPasswordHash()`. Secondly, the **authorization object** returned will be available to any web methods made if authentication is successful. It is implementation-specific. But it provides a mechanism whereby web methods can retrieve information about the user currently authenticated. We will not use this feature, so we only return the username. But it could be a pointer to the user data record or similar information as well.

Running the application now and trying to access any of the data export resources from a browser will prompt the user with a user login window. To get the data, enter the username `Admin` and password `Password` to continue.

Web form login

We have now secured our machine-to-machine (M2M) interfaces. We now want to secure our human-to-machine (H2M) interfaces. This is preferably done using a web form login procedure and sessions, as described in the introduction to the HTTP protocol.

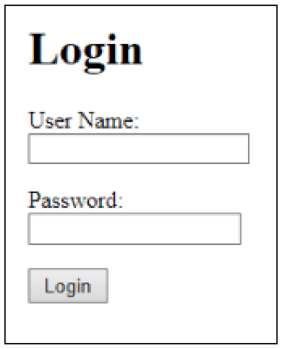
The first thing we need to do is to update the generation of our root web page, the contents in the try clause in our `HttpGetRoot()` method, as follows. Here, we need to add the following highlighted code:

```
string SessionId = req.Header.GetCookie ("SessionId");

resp.ContentType = "text/html";
resp.Encoding = System.Text.Encoding.UTF8;
resp.ReturnCode = HttpStatusCode.Successful_OK;

if (CheckSession (SessionId))
{
  resp.Write ("<html><head><title>Sensor</title></head>");
  resp.Write ("<body><h1>Welcome to Sensor</h1>");
  resp.Write ("<p>Below, choose what you want to do.</p><ul>");
  resp.Write ("<li><a href='/credentials'>");
  resp.Write ("Update login credentials.</a></li>");
  resp.Write ("<li>View Data</li><ul>");
  resp.Write ("<li><a href='/xml?Momentary=1'>");
  resp.Write ("View data as XML using REST</a></li>");
  resp.Write ("<li><a href='/json?Momentary=1'>");
  resp.Write ("View data as JSON using REST</a></li>");
  resp.Write ("<li><a href='/turtle?Momentary=1'>");
  resp.Write ("View data as TURTLE using REST</a></li>");
  resp.Write ("<li><a href='/rdf?Momentary=1'>");
  resp.Write ("View data as RDF using REST</a></li>");
  resp.Write ("<li><a href='/html'>");
  resp.Write ("Data in a HTML page with graphs</a></li></ul>");
  resp.Write ("</body></html>");
} else
  OutputLoginForm (resp, string.Empty);
```

We first check whether there is a cookie for us, with the identity of `SessionId`. This cookie is site-specific, so we don't need to worry that it will be confused with cookies from other sites. We then call the `CheckSession()` method to make sure the session is a valid session, and not a fabricated or old session identity. If the session is valid, we continue by displaying the menu as we did before, with one additional item; a link to a page with the `/credentials` path, where the user can change their username and password, if desired. If the session is not valid, the `OutputLoginForm()` method is called. This method generates a user login form, together with an optional message to the user, as follows:

```
private static void OutputLoginForm (HttpServerResponse resp,
  string Message)
{
  resp.Write ("<html><head><title>Sensor</title></head>");
  resp.Write ("<body><form method='POST' action='/' ");
  resp.Write ("target='_self' autocomplete='true'>");
  resp.Write (Message);
  resp.Write ("<h1>Login</h1>");
  resp.Write ("<p><label for='UserName'>");
  resp.Write ("User Name:</label><br/>");
  resp.Write ("<input type='text' name='UserName'/></p>");
  resp.Write ("<p><label for='Password'>");
  resp.Write ("Password:</label><br/>");
  resp.Write ("<input type='password' name='Password'/></p>");
  resp.Write ("<p><input type='submit' value='Login'/></p>");
  resp.Write ("</form></body></html>");
}
```

This little form will look as follows when viewed in a browser:



Receiving a POST request

When the user enters their credentials and clicks the **Login** button, an HTTP POST message is sent to the resource specified in the `action` attribute in the `form` tag. In our case, the form is posted back to the / resource. So we need to add a POST web handler when we register the resources, as follows:

```
HttpServer.Register ("/", HttpGetRoot, HttpPostRoot, false);
```

The basic structure of this web resource handler is the same as the previous ones, and shouldn't need much explanation at this point:

```
private static void HttpPostRoot (HttpServerResponse resp,
  HttpServerRequest req)
{
  networkLed.High ();
  try
  {
  }
  finally
  {
    networkLed.Low ();
  }
}
```

The major difference between a GET and POST operation is that the client encodes data with the POST operation. The HTTP server decodes this data using information available in the request and available MIME decoders, and makes the decoded object available in the `Data` property of the request object. We expect a `FormParameters` object, containing our form parameters, and anything else should return a bad request error message back to the client. We implement this as follows, all within the try section in the preceding code snippet:

```
FormParameters Parameters = req.Data as FormParameters;
if (Parameters == null)
  throw new HttpException (
    HttpStatusCode.ClientError_BadRequest);
```

Now that we know a form has been posted, we extract the username and password from the form, and see whether any such user is known by the system:

```
string UserName = Parameters ["UserName"];
string Password = Parameters ["Password"];
string Hash;
object AuthorizationObject;
```

```
GetDigestUserPasswordHash (UserName,
  out Hash, out  AuthorizationObject);
```

If the user does not exist, the hash and authorization object will be `null`. If the user exists, but the stored hash is different from the hash you get using the provided username and password, you know that the password provided is wrong. Regardless of whether it is the user name that is wrong or the password that is wrong, (so that the client cannot deduce whether a username exists in the system or not), you return the same error message back to the client:

```
if (AuthorizationObject == null ||
  Hash != CalcHash (UserName, Password))
{
  resp.ContentType = "text/html";
  resp.Encoding = System.Text.Encoding.UTF8;
  resp.ReturnCode = HttpStatusCode.Successful_OK;

  Log.Warning ("Invalid login attempt.", EventLevel.Minor,
    UserName, req.ClientAddress);
  OutputLoginForm (resp, "<p>The login was incorrect. " +
    "Either the user name or the password was " +
    "incorrect. Please try again.</p>");
```

Note that we always log an invalid login attempt. This makes it possible, at a later stage, to build in monitors that monitor invalid login attempts in the network, to detect malicious intrusion attempts.

Improving navigation experience using PRG

If the login attempt was successful, we log that information too, but we also create a session for the user and return it in a cookie back to the client. The client will send this cookie to the server each time a request is made.

One thing is important to note in the following code: instead of returning a successful operation (200) and a corresponding page, we return a `See Other` redirection status code, and point the browser back to the same resource. This response code will trigger the browser to reload the page, but using GET instead. Now that the user is logged in, the session will be detected and the proper page displayed. This communication pattern is called the PRG pattern, or POST/Redirect/GET pattern.

It is often used to avoid navigation problems in browsers, where users use the back and forward buttons to go back and forth between pages. By using the `See Other` response code, the server makes sure the POST operation is removed from the history of the browser and is replaced by a GET to the same resource. This prevents the user being bombarded with questions if they want the browser to re-send information back to the server, something that would also confuse the server. Our implementation of all that we just discussed is this:

```
} else
{
  Log.Information ("User logged in.", EventLevel.Minor,
    UserName, req.ClientAddress);

  string SessionId = CreateSessionId (UserName);
  resp.SetCookie ("SessionId", SessionId, "/");
  resp.ReturnCode = HttpStatusCode.Redirection_SeeOther;
  resp.AddHeader ("Location", "/");
  resp.SendResponse ();
}
```

Creating sessions

When we create sessions, we need a data structure that allows us to quickly find a session, provided its identity, but also a structure that allows us to remove old unused sessions. For this reason, we create two static dictionaries, one to look up the last time a session was referenced and by whom, and another sorted dictionary that can be used to look up the session ID given the last time it was referenced:

```
private static Dictionary<string,
  KeyValuePair<DateTime, string>> lastAccessBySessionId =
    new Dictionary<string, KeyValuePair<DateTime, string>> ();

private static SortedDictionary<DateTime,string>
  sessionIdByLastAccess =
    new SortedDictionary<DateTime, string> ();
```

We will also need a well-defined session timeout (which we set to two minutes) and a random number generator:

```
private static readonly TimeSpan sessionTimeout =
  new TimeSpan (0, 2, 0);
private static Random gen = new Random ();
```

The creation of the session identity is easy. It has to be sufficiently complex so that a client cannot guess it using a reasonable amount of effort. For our purposes, it is sufficient to use a GUID. We create it as follows:

```
private static string CreateSessionId (string UserName)
{
  string SessionId = Guid.NewGuid ().ToString ();
  return SessionId;
}
```

Before returning the session identity to the caller, we must insert it into our data structure defined previously. We do this in the following way. We note here that it is theoretically possible that two sessions are created simultaneously. Since one of the dictionaries uses a `DateTime`-valued key, we need to check for this eventuality. If this happens, we add a small random number amount of ticks to the timestamp of the session ID, until no such key is used. For all practical purposes, this is sufficient. We also need to execute any code accessing the data structures within a critical section to make sure only one thread at a time accesses the data structure. We do this using the `lock` statement:

```
DateTime Now = DateTime.Now;

lock (lastAccessBySessionId)
{
  while (sessionIdByLastAccess.ContainsKey (Now))
    Now = Now.AddTicks (gen.Next (1, 10));

  sessionIdByLastAccess [Now] = SessionId;
  lastAccessBySessionId [SessionId] =
    new KeyValuePair<DateTime, string> (Now, UserName);
}
```

Validating a session

Each time a request is made to a web resource, referencing a session, the web resource needs to validate the session. This means the application needs to make sure that the session exists, it is not too old, but also update the last access timestamp of the session to make sure it is kept alive as long as it is used.

To validate a session, we create the following method that returns `true` if a session exists and is valid and `false` if it does not exist or is invalid. The method also returns the name of the user last using the session, if found:

```
internal static bool CheckSession (string SessionId,
  out string UserName)
```

```
{
  KeyValuePair<DateTime, string> Pair;
  DateTime TP;
  DateTime Now;

  UserName = null;
```

The first obvious check is to see whether the session identity is found in the data structure. Make sure that all access to the session data structures are made in a critical section, to make sure data is not corrupted if multiple threads access it simultaneously:

```
  lock (lastAccessBySessionId)
  {
    if (!lastAccessBySessionId.TryGetValue (SessionId,
      out Pair))
        return false;
```

The next check is to make sure the session found in memory is not an old one that should have been discarded:

```
    TP = Pair.Key;
    Now = DateTime.Now;

    if (Now - TP > sessionTimeout)
    {
      lastAccessBySessionId.Remove (SessionId);
      sessionIdByLastAccess.Remove (TP);
      return false;
    }
```

If the session is valid, we update the information about the session to make sure it is remembered by the current timestamp. Care has to be taken to make sure it does not collide with another session:

```
    sessionIdByLastAccess.Remove (TP);
    while (sessionIdByLastAccess.ContainsKey (Now))
      Now = Now.AddTicks (gen.Next (1, 10));

    sessionIdByLastAccess [Now] = SessionId;
    UserName = Pair.Value;
    lastAccessBySessionId [SessionId] =
      new KeyValuePair<DateTime, string> (Now,
         UserName);
  }

  return true;
}
```

If we are not interested in the user name of a given session, and only want to validate the session itself, we add the following simple version of the same method:

```
private static bool CheckSession (string SessionId)
{
  string UserName;
  return CheckSession (SessionId, out UserName);
}
```

Now that we have a way to check the current session, we need to make sure it is used in all our web services that are not protected by WWW-authentication. We add the following code to each such web handler. If somebody tries to access one of these resources and the user is not using a valid session, the user is negated access to the resource and redirected to the root resource where the login-form will be displayed:

```
string SessionId = req.Header.GetCookie ("SessionId");
if (!CheckSession (SessionId))
  throw new HttpTemporaryRedirectException ("/");
```

Note that if a call does not have the cookie, the `GetCookie()` method returns the empty string, which will always make the `CheckSession()` method fail. Note that we do not add this code to the resources with WWW-authentication enabled, since these must be possible to reach by machines, not logging in to the web application, thus not creating sessions either.

Removing old sessions

To avoid memory leaks, the application must remove old sessions from memory. We can use the fact that we have a sorted dictionary where session identities are sorted by last access time to understand that the first entries will be the oldest entries. We can also use the fact that it is very seldom an entry needs to be removed to create a method that begins by looping through available records, and breaking the loop when entries are found that are newer than the limit required for removal. We cannot remove entries while looping since that would destroy the enumerator of the dictionary, so we need to store them in a temporary structure and remove them later. We begin with declaring the method:

```
private static void RemoveOldSessions ()
{
  Dictionary<string,KeyValuePair<DateTime, string>>
    ToRemove = null;
  DateTime OlderThan = DateTime.Now.Subtract (sessionTimeout);
  KeyValuePair<DateTime, string> Pair2;
  string UserName;
```

```
    lock (lastAccessBySessionId)
    {
      foreach (KeyValuePair<DateTime,string>Pair
        in sessionIdByLastAccess)
      {
        if (Pair.Key <= OlderThan)
        {
          if (ToRemove == null)
            ToRemove = new Dictionary<string,
              KeyValuePair<DateTime, string>>();

          if (lastAccessBySessionId.TryGetValue (
            Pair.Value, out Pair2))
              UserName = Pair2.Value;
          else
            UserName = string.Empty;

          ToRemove [Pair.Value] =
            new KeyValuePair<DateTime, string>
              (Pair.Key, UserName);
        } else
          break;
      }
```

Now that we have identified the sessions to remove, we can safely remove them without destroying enumerators. We also make sure we log information about the removal of a session, so that we can match each login with when the corresponding session ends:

```
      if (ToRemove != null)
      {
        foreach (KeyValuePair<string,
          KeyValuePair<DateTime, string>>Pair in ToRemove)
        {
          lastAccessBySessionId.Remove (Pair.Key);
          sessionIdByLastAccess.Remove
            (Pair.Value.Key);

          Log.Information ("User session closed.",
             EventLevel.Minor, Pair.Value.Value);
        }
      }
    }
  }
```

From where do we call the `RemoveOldSessions()` method? It has to be called regularly. So the best place is at the end of the measurement timer, after turning the measurement LED off:

```
} finally
{
  measurementLed.Low ();
  RemoveOldSessions ();
}
```

Overriding WWW-authentication

Running the application now will allow the user to log in and navigate through its pages. But when navigating to any of the sensor data export pages, a second login dialog will appear, requesting user credentials to get access. This is because we have said these resources are protected by the WWW-authentication mechanism built into the HTTP protocol itself. To get access to these resources if you're logged in to the web page and browse the resources using from a session, you need to override the standard WWW-authentication mechanism provided by the HTTP protocol. Luckily, this is not difficult to do. What we need to do is simply create a new WWW-authentication mechanism, a mechanism that doesn't send any challenge to the client, and accepts as the authenticated user, if a valid session identity is provided in the request header's cookies. We begin by defining the authentication mechanism as follows:

```
public class SessionAuthentication :
  HttpServerAuthenticationMethod
{
  public SessionAuthentication ()
  {
  }
}
```

To make sure a challenge is not sent to the client, we return an empty challenge as follows:

```
public override string Challenge
{
  get
  {
    return string.Empty;
  }
}
```

Then we authenticate any user with a valid session identity provided as a cookie:

```
public override object Authorize (HttpHeader Header,
  HttpServer.Method Method, IHttpServerResource Resource,
  System.Net.EndPoint RemoteEndPoint, out string UserName,
  out UnauthorizedReason Reason)
{
  string SessionId = Header.GetCookie ("SessionId");

  if (MainClass.CheckSession (SessionId, out UserName))
  {
    Reason = UnauthorizedReason.NoError;
    return UserName;
  }
  else
  {
    Reason = UnauthorizedReason.OldCredentialsTryAgain;
    return null;
  }
}
```

We also need to register the new authentication method with the HTTP server during application initialization. This is done in the same way as we registered the digest authentication method, as follows:

```
HttpServer.RegisterAuthenticationMethod (
  new SessionAuthentication ());
```

Now when navigating to these resources from within a session, no additional WWW-authentication is required. But anybody trying to access these resources outside of a session will be prompted with a WWW-authentication login dialog box.

Editing user credentials

To complete the authentication feature, we must allow the user or owner to edit the user credentials, so that they can be personalized. We've already put a link to a `/credentials` resource on our first page. We now register this resource, as follows:

```
HttpServer.Register ("/credentials", HttpGetCredentials,
  HttpPostCredentials, false);
```

The **`HttpGetCredentials()`** method checks the session and returns the credential forms, generated by **`OutputCredentialsForm()`**. We put the form in a separate method, so that we can call it from the POST method handler as well:

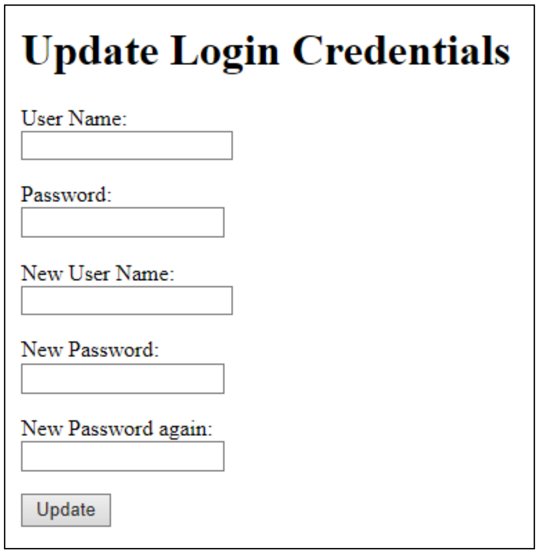
```
private static void HttpGetCredentials (HttpServerResponse resp,
  HttpServerRequest req)
{
  networkLed.High ();
  try
  {
    string SessionId = req.Header.GetCookie ("SessionId");
    if (!CheckSession (SessionId))
      throw new HttpTemporaryRedirectException ("/");

    resp.ContentType = "text/html";
    resp.Encoding = System.Text.Encoding.UTF8;
    resp.ReturnCode = HttpStatusCode.Successful_OK;

    OutputCredentialsForm (resp, string.Empty);
  } finally
  {
    networkLed.Low ();
  }
}
```

The generation of the credentials form is also straightforward, repeating what we did with earlier forms, except we now add a few more input fields:

```
private static void OutputCredentialsForm (
  HttpServerResponse resp, string Message)
{
  resp.Write ("<html><head><title>Sensor</title></head><body>");
  resp.Write ("<form method='POST' action='/credentials' ");
  resp.Write ("target='_self' autocomplete='true'>");
  resp.Write (Message);
  resp.Write ("<h1>Update Login Credentials</h1>");
  resp.Write ("<p><label for='UserName'>");
  resp.Write ("User Name:</label><br/>");
  resp.Write ("<input type='text' name='UserName'/></p>");
  resp.Write ("<p><label for='Password'>");
  resp.Write ("Password:</label><br/>");
  resp.Write ("<input type='password' name='Password'/></p>");
  resp.Write ("<p><label for='NewUserName'>");
  resp.Write ("New User Name:</label><br/>");
  resp.Write ("<input type='text' name='NewUserName'/></p>");
  resp.Write ("<p><label for='NewPassword1'>");
  resp.Write ("New Password:</label><br/>");
  resp.Write ("<input type='password' ");
```

```
    resp.Write ("name='NewPassword1'/></p>");
    resp.Write ("<p><label for='NewPassword2'>");
    resp.Write ("New Password again:</label><br/>");
    resp.Write ("<input type='password' ");
    resp.Write ("name='NewPassword2'/></p>");
    resp.Write ("<p><input type='submit' value='Update'/></p>");
    resp.Write ("</form></body></html>");
  }
```

When viewing the form in a browser, it will look as follows:


Update Login Credentials

User Name:

Password:

New User Name:

New Password:

New Password again:

Updating user credentials

When a user presses the **Update** button a POST message is sent to the same resource `/credentials` path. This is defined in the `form` tag of the HTML page. We've already registered the method handler. We create the method in the usual manner:

```
private static void HttpPostCredentials (HttpServerResponse resp,
  HttpServerRequest req)
{
  networkLed.High ();
  try
  {
    string SessionId = req.Header.GetCookie ("SessionId");
    if (!CheckSession (SessionId))
      throw new HttpTemporaryRedirectException ("/");
  } finally
  {
    networkLed.Low ();
  }
}
```

We expect the POST request is made together with a set of form parameters. If not we consider it a bad request:

```
    FormParameters Parameters = req.Data as FormParameters;
    if (Parameters == null)
      throw new HttpException (
        HttpStatusCode.ClientError_BadRequest);
```

All the following validations will, if they fail, return messages in HTML to the user. So, on the HTTP level, these errors are still considered OK. We therefore tell the client the type of content being returned and that the operation went well:

```
    resp.ContentType = "text/html";
    resp.Encoding = System.Text.Encoding.UTF8;
    resp.ReturnCode = HttpStatusCode.Successful_OK;
```

We extract the parameters from the form first. We don't have to check whether the parameters exist or not. If they don't exist in the form, the empty string will be returned for the corresponding parameter:

```
    string UserName = Parameters ["UserName"];
    string Password = Parameters ["Password"];
    string NewUserName = Parameters ["NewUserName"];
    string NewPassword1 = Parameters ["NewPassword1"];
    string NewPassword2 = Parameters ["NewPassword2"];
```

We must look for the current user credentials again, to make sure the correct user hasn't left their computer with the browser pointing to our device, and somebody else passes by and changes the credentials, just for the fun of it:

```
    string Hash;
    object AuthorizationObject;

    GetDigestUserPasswordHash (UserName, out Hash,
      out  AuthorizationObject);
```

If the credentials provided are not correct, we return the form again, with an appropriate error message to the user with the help of the following code. We also log the invalid attempt to the event log, so that we later can detect intrusion attempts in the network (if we want to):

```
if (AuthorizationObject == null ||
  Hash != CalcHash (UserName, Password))
{
  Log.Warning ("Invalid attempt to change login credentials.",
    EventLevel.Minor, UserName, req.ClientAddress);
  OutputCredentialsForm (resp, "<p>Login credentials " +
    "provided were not correct. Please try again.</p>");
```

A common mistake is to misspell a password. When changing the credentials, this can be fatal, since you might not figure out how you misspelled the password. For this reason, it is always important to provide two password fields and make sure that they are equal. If not, an appropriate error message must be returned to the user:

```
} else if (NewPassword1 != NewPassword2)
{
  OutputCredentialsForm (resp,
    "<p>The new password was not entered correctly. " +
    "Please provide the same new password twice.</p>");
```

Some form of limit on username and password complexity is also a good idea. In our example, we will be satisfied if the username and password are not empty:

```
} else if (string.IsNullOrEmpty (UserName) ||
  string.IsNullOrEmpty (NewPassword1))
{
  OutputCredentialsForm (resp, "<p>Please provide a " +
    "non-empty user name and password.</p>");
```

We must also make sure that somebody does not try to break the application by sending a very long username. The length of the password is not important, since we compute a hash of it anyway:

```
} else if (UserName.Length > DB.ShortStringClipLength)
{
  OutputCredentialsForm (resp, "<p>The new user name " +
    "was too long.</p>");
```

If all is well and the new credentials pass all tests, we log this fact and update the `credentials` object. Note that our implementation of `LoginCredentials` is such that the `Modified` property only becomes true if we actually set a new value to any of the properties. This means the object will only be saved to the database, if actually saved. After saving the object, we use the *PRG pattern* described earlier to redirect the user to the main page instead of returning a success response. This prevents problems in the browser if navigating back and forth in the navigator history. This is demonstrated by the following code snippet:

```
} else
{
  Log.Information ("Login credentials changed.",
    EventLevel.Minor, UserName, req.ClientAddress);

  credentials.UserName = NewUserName;
  credentials.PasswordHash =
    CalcHash (NewUserName, NewPassword1);
  credentials.UpdateIfModified ();

  resp.ReturnCode = HttpStatusCode.Redirection_SeeOther;
  resp.AddHeader ("Location", "/");
  resp.SendResponse ();
}
```

H
Delayed Responses in HTTP

There are various ways to send events using HTTP — for instance, our sensor project implements an HTTP server on a sensor. One way to send events is to delay the response to a specific type of request, and only send the response when something interesting happens or when the request times-out. This solution has the benefit that it can be used to send events to clients not directly reachable from the sensor. We will dedicate this appendix to showing how we can inform interested parties of events that occur on the device when they occur, without the need for constant polling of the device. This architecture will lend itself naturally to a subscription pattern, where different parties can subscribe to different types of events in a natural fashion. It builds on the sensor project presented in the book.

Pending events

Firstly, we must recognize that an HTTP server cannot send data to anybody without a previous request. So, in our case, an entity interested in certain types of events on the sensor must first make a request to the device. This request will act as a subscription to the events concerned.

Normally, the HTTP server will respond immediately when processing the request. But, for these event subscriptions, the server will not respond. Instead, it will put the events in a list of pending event subscriptions. When an event occurs, the device looks through the list of pending event subscriptions to see if the event matches what is looked for. If so, a response is formed and returned to the client requesting the information.

The time between the request and the response can theoretically be very long. But the TCP layer in the network will drop a connection if no communication is done over it in a given amount of time. This time may vary across networks. We need to consider this and introduce a timeout parameter in the request. If no event occurs during this time, a response must be returned anyway, so as to not lose the TCP connection on which the HTTP protocol works.

Now, anybody interested in events from the sensor has an efficient means of achieving this and does not need to poll the device very quickly in order to have updated information. It sends an event subscription request to the sensor, and waits for a response. As soon as it receives a response, it sends a new request and waits for its response. . . And so on. As soon as events that the interested party is interested in occur, they will be sent. Events of no interest will not be sent. In this way, communication is very efficient, and allows for entities to subscribe to different types of information.

Defining a pending event

We need a data representation in our sensor of what a pending event is. We will create four types of events:

- If the temperature value differs from a given set value by more than a specified amount
- If the light value differs from a given set value by more than a specified amount
- If the motion value differs from a given set value
- If nothing happens for a given number of seconds

We recognize that an entity may be interested in a combination of the above. So we create the following class to represent what an entity is interested in:

```
public class PendingEvent
{
  private HttpServerResponse response;
  private ISensorDataExport exportModule;
  private string contentType;
  private double? temp = null;
  private double? tempDiff = null;
  private double? light = null;
  private double? lightDiff = null;
  private bool? motion = null;
  private DateTime timeout;
```

We need to remember the HTTP response object to send any responses to, as well as a sensor data export module and its corresponding content type, representing the format in which to send the data when the event occurs. We also add a `timeout` parameter that represents the latest point in time at which a response has to be returned to the subscriber, regardless of events or not.

We then use nullable data types (using the question mark) to represent what an entity is interested in. If the corresponding value is `null`, the entity is not interested in the corresponding field. If it is not null, it represents an interest. For temperature, two parameters define the event. First `temp`, which corresponds to the temperature the subscriber has, and then `tempDiff`, which corresponds to the maximum allowed difference between the actual temperature in the sensor and the value available in `temp`, before the sensor needs to send an event back to the subscriber. Light is handled in the same way, using the two nullable member variables: `light` and `lightDiff`. Since the motion detector is binary, it is sufficient to have only one nullable member variable motion, representing the value known to the subscriber, if present. If the actual value differs from this, an event has to be returned to the subscriber.

We create a constructor containing all the necessary information, as follows:

```
public PendingEvent (double? Temp, double? TempDiff,
  double? Light, double? LightDiff, bool? Motion, int Timeout,
  HttpServerResponse Response, string ContentType,
  ISensorDataExport ExportModule)
{
  this.temp = Temp;
  this.tempDiff = TempDiff;

  this.light = Light;
  this.lightDiff = LightDiff;

  this.motion = Motion;

  this.timeout = DateTime.Now.AddSeconds (Timeout);
  this.response = Response;
  this.contentType = ContentType;
  this.exportModule = ExportModule;
}
```

We also need to publish some of the properties for later use, when the application sends the event back to the subscriber:

```
public HttpServerResponse Response
{
  get{ return this.response; }
}

public string ContentType
{
  get{ return this.contentType; }
}

public ISensorDataExport ExportModule
{
  get{ return this.exportModule; }
}
```

Triggering an event

We also add a method that sends the current values to the object; the object returns if the event is triggered or not:

```
public bool Trigger (double Temp, double Light, bool Motion)
{
  if (this.motion.HasValue && this.motion.Value ^ Motion)
    return true;

  if (this.temp.HasValue && this.tempDiff.HasValue &&
    Math.Abs (this.temp.Value-Temp) >= this.tempDiff.Value)
      return true;

  if (this.light.HasValue && this.lightDiff.HasValue &&
    Math.Abs (this.light.Value - Light) >=
    this.lightDiff.Value)
      return true;

  if (DateTime.Now >= this.timeout)
    return true;

  return false;
}
```

Registering event subscription resources

In the main application, we register four asynchronous web resources that entities can use to subscribe to events, one web resource for each data format we support. Remember that synchronous web resources are responded to within the context of a working thread maintained by the HTTP server, and that asynchronous web resources are responded to outside the context of the HTTP server. Since we will not respond to subscription requests right away, we need to define them as asynchronous web resources (the fourth parameter control if they are synchronous, the third if they require www-authentication):

```
HttpServer.Register ("/event/xml",
  HttpGetEventXml, true, false);
HttpServer.Register ("/event/json",
  HttpGetEventJson, true, false);
HttpServer.Register ("/event/turtle",
  HttpGetEventTurtle, true, false);
HttpServer.Register ("/event/rdf",
  HttpGetEventRdf, true, false);
```

Publishing event subscription resources

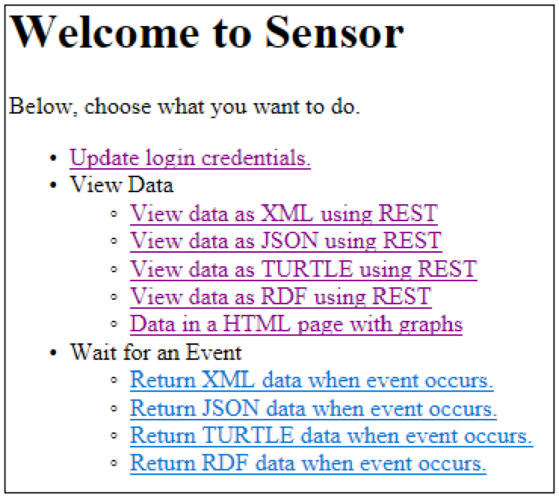
We want to add links to our event subscription resources on our root page, so we can test it. In our `HttpGetRoot()` method, after checking the session, we first construct a set of event subscription parameters representing the current state of the sensor, and the fact that we are interested in any change more than one unit from the current value:

```
StringBuilder sb = new StringBuilder ();
string EventParameters;

lock (synchObject)
{
  sb.Append ("?Temperature=");
  sb.Append (XmlUtilities.DoubleToString (temperatureC, 1));
  sb.Append ("&amp;TemperatureDiff=1&amp;Light=");
  sb.Append (XmlUtilities.DoubleToString (lightPercent, 1));
  sb.Append ("&amp;LightDiff=10&amp;Motion=");
  sb.Append (motionDetected ? "1" : "0");
  sb.Append ("&amp;Timeout=25");
}

EventParameters = sb.ToString ();
```

We then insert the new links to our page at the end:

```
resp.Write ("<li>Wait for an Event</li><ul>");
resp.Write ("<li><a href='/event/xml");
resp.Write (EventParameters);
resp.Write ("'>Return XML data when event occurs.</a></li>");
resp.Write ("<li><a href='/event/json");
resp.Write (EventParameters);
resp.Write ("'>Return JSON data when event occurs.</a></li>");
resp.Write ("<li><a href='/event/turtle");
resp.Write (EventParameters);
resp.Write ("'>Return TURTLE data when event occurs.</a></li>");
resp.Write ("<li><a href='/event/rdf");
resp.Write (EventParameters);
resp.Write ("'>Return RDF data when event occurs.</a></li>");
resp.Write ("</ul></body></html>");
```

Our root page will now look as follows:



Welcome to Sensor

Below, choose what you want to do.

- Update login credentials.
- View Data
  - View data as XML using REST
  - View data as JSON using REST
  - View data as TURTLE using REST
  - View data as RDF using REST
  - Data in a HTML page with graphs
- Wait for an Event
  - Return XML data when event occurs.
  - Return JSON data when event occurs.
  - Return TURTLE data when event occurs.
  - Return RDF data when event occurs.

Adding event subscription method handlers

In the main application, we define a list of pending events that entities can subscribe to, as follows:

```
private static List<PendingEvent> pendingEvents =
  new List<PendingEvent> ();
```

For each data format supported, we define the method handlers as follows. Each method handler simply calls a generic method handler that registers the corresponding event subscription in the list of pending events. Each method also provides the generic method with the corresponding sensor data export module required to export the data when the event occurs, as well as the corresponding content type:

```
private static void HttpGetEventXml (HttpServerResponse resp,
  HttpServerRequest req)
{
  HttpGetEvent (resp, req, "text/xml",
    new SensorDataXmlExport (resp.TextWriter));
}

private static void HttpGetEventJson (HttpServerResponse resp,
  HttpServerRequest req)
{
  HttpGetEvent (resp, req, "application/json",
    new SensorDataJsonExport (resp.TextWriter));
}

private static void HttpGetEventTurtle (HttpServerResponse resp,
  HttpServerRequest req)
{
  HttpGetEvent (resp, req, "text/turtle",
    new SensorDataTurtleExport (resp.TextWriter, req));
}

private static void HttpGetEventRdf (HttpServerResponse resp,
  HttpServerRequest req)
{
  HttpGetEvent (resp, req, "application/rdf+xml",
    new SensorDataRdfExport (resp.TextWriter, req));
}
```

The structure of the generic method is similar to previous web methods we've implemented:

```
private static void HttpGetEvent (HttpServerResponse resp,
  HttpServerRequest req, string ContentType,
  ISensorDataExport ExportModule)
{
  networkLed.High ();
  try
  {
  }
  finally
  {
    networkLed.Low ();
  }
}
```

Parsing event query parameters

In the try-clause, we start parsing the query parameters. We begin with the required `Timeout` parameter, stating the maximum number of seconds to wait for an event to occur before returning a response:

```
int Timeout;
if (!req.Query.TryGetValue ("Timeout", out s) ||
  !int.TryParse (s, out Timeout) || Timeout <= 0)
    throw new HttpException (
      HttpStatusCode.ClientError_BadRequest);
```

We then parse the optional `Temperature` and `TemperatureDiff` query parameters. Both are required to exist and be valid for us to accept the parameters in the event subscription:

```
double? Temperature = null;
double? TemperatureDiff = null;
double d, d2;
string s;
if (req.Query.TryGetValue ("Temperature", out s) &&
  XmlUtilities.TryParseDouble (s, out d) &&
  req.Query.TryGetValue ("TemperatureDiff", out s) &&
  XmlUtilities.TryParseDouble (s, out d2) && d2 > 0)
{
  Temperature = d;
  TemperatureDiff = d2;
}
```

Similarly, we parse the optional `Light` and `LightDiff` query parameters, and if they exist and are valid, we include them in the event subscription:

```
double? Light = null;
double? LightDiff = null;
if (req.Query.TryGetValue ("Light", out s) &&
  XmlUtilities.TryParseDouble (s, out d) &&
  req.Query.TryGetValue ("LightDiff", out s) &&
  XmlUtilities.TryParseDouble (s, out d2) && d2 > 0)
{
  Light = d;
  LightDiff = d2;
}
```

Parsing the optional `Motion` query parameter is simpler, since it is only one Boolean parameter:

```
bool? Motion = null;
bool b;
if (req.Query.TryGetValue ("Motion", out s) &&
  XmlUtilities.TryParseBoolean (s, out b))
    Motion = b;
```

If the event subscription doesn't contain any valid event subscription information, we return a bad request error immediately. It is unnecessary to add the pending event, simply to wait for it to time out:

```
if (!(Temperature.HasValue || Light.HasValue || Motion.HasValue))
  throw new HttpException (
    HttpStatusCode.ClientError_BadRequest);
```

Finally, we add the parsed event subscription to our list of pending events:

```
lock (synchObject)
{
  pendingEvents.Add (new PendingEvent (Temperature,
    TemperatureDiff, Light, LightDiff, Motion, Timeout,
    resp, ContentType, ExportModule));
}
```

Checking for new events

We are now ready to check for new events. Our `SampleSensorValues()` method, called regularly to sample new values, is the ideal place to check for new events, since new measurements are fresh off the press. Still within the critical section, before releasing the locked synchronization object, we loop through any pending events and verify whether they are triggered by the new measured sensor values. If the event is triggered, a corresponding response containing all the sensor's momentary values is returned immediately to the subscriber and the event is removed from the list.

```
PendingEvent Event;
int i = 0;
int c = pendingEvents.Count;

while (i < c)
{
  Event = pendingEvents [i];

  if (Event.Trigger (temperatureC,
    lightPercent, motionDetected))
  {
    pendingEvents.RemoveAt (i);
    c--;

    HttpGetSensorData (Event.Response,
      Event.ContentType, Event.ExportModule,
      new ReadoutRequest (ReadoutType.MomentaryValues));
    Event.Response.SendResponse ();

    } else
    i++;
}
```

Note here that it is not sufficient to only export the sensor data to the response object. Since we are outside the context of the HTTP server, we need to say that the response is complete and also that we do not intend to write anything else to the client. We do this by calling the `SendResponse()` method on the `Response` object.

Testing event subscriptions

We are now ready to test our event subscription functionality. For instance, the following URL will wait either until 25 seconds have passed, or the light density changes from 81.3 percent by at least 5 units of a percent. The response would be all momentary values of the sensor, in XML format at `http://192.168.0.29/event/xml?Light=81.3&LightDiff=5&Timeout=25`.

The following URL would wait until 25 seconds have passed, or motion is detected before this. The response is returned in JSON format at `http://192.168.0.29/event/xml?Motion=0&Timeout=25`.

Several fields can be monitored at the same time. For instance, the following URL will wait a maximum of 25 seconds for a change in temperature of 1 degree from 21.2 degrees, a change in light density of more than 5 units of a percent from 81.3 percent, or for the motion to stop. Results are returned in XML at `http://192.168.0.29/event/xml?Temperature=21.2&TemperatureDiff=1&Light=81.3&LightDiff=5&Motion=1&Timeout=25`.

Why 25 seconds? Here we assume there is a risk that the TCP connection may be dropped after 30 seconds of inactivity. Having a timeout of 25 seconds, gives us a 5-second margin before such an event.

I
Fundamentals of UPnP

As soon as HTTP gained popularity, the protocol started to be used in ways not originally conceived of. Instead of transferring hypertext documents with embedded content from servers to clients viewing them, the HTTP protocol became a popular tool for machines calling services on other machines with what is known today as web services. But HTTP as it was defined, even with technologies such as web services, did not provide sufficient tools for solutions in ad hoc networks. (Ad hoc networks are networks with no predefined topology or configuration, where devices adapt themselves to the surrounding environment.) One attempt to solve this problem was the initiative that is now known as **Universal Plug and Play** (**UPnP** for short). It has become the standard protocol for consumer electronics; it adapts itself to home or office networks with almost no manual configuration. This appendix will introduce the UPnP protocol to the reader and show how it can be used by things to automatically connect to equipment in its surroundings and how their existing interfaces can be utilized in distributed applications. It will also discuss current extensions of UPnP into the Internet of Things and the Cloud.

Extending HTTP

For many reasons, HTTP quickly became very popular. These reasons include ease of use through standard software such as browsers, versatility in transporting content through the use of internet media types, and a simple way to identify resources in a distributed network using URLs and so on. This made HTTP very easy to extend from its initial purpose. It is not strange that HTTP was used as a means to achieve machine-to-machine communication in networks, regardless of its limitations; if something works, you take that and build what is missing on top.

Finding resources in the network

There are various ways to make things interact in a network using HTTP, as we saw in *Chapter 2*, *The HTTP Protocol*. But how do things find each other in the first place? In ad hoc networks, participants do not know beforehand what resources exist. How does the computer find a printer in the network? Or how does a security system find available cameras in the network? Before any meaningful conversation can take place, participants in the conversation need to find each other. Early solutions relied on manual configuration, where a human operator configured each device or machine in the network so that they could talk to each other. How can such configuration be done automatically?

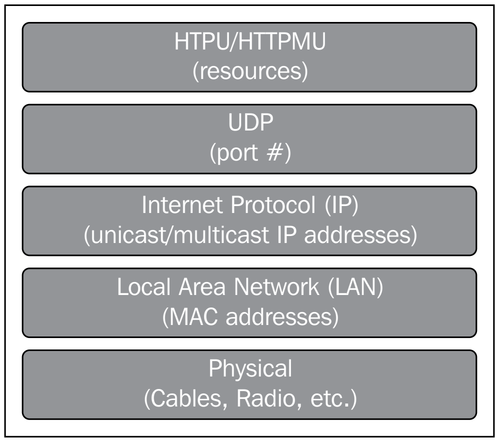
Finding content quickly became a big problem for the evolving web. Most solutions were based on active searching where search engines "crawl" the web, first searching for computers using HTTP and then trying to figure out what content each and every one hosted. This was mainly done by following links in hypertext documents, but it can also be done by testing for different common patterns. This solution works relatively well for human-readable content based on pages published on the web, where the authors have an interest in publishing and referencing content, and therefore makes sure it is registered correctly. But, for new devices in a network, this is a poor solution. First of all, there will be no initial page with an initial link pointing to the newly installed device in the network. Secondly, most things do not host human-readable content based on pages, as such, but publish services. To figure out what services exist and how they work simply using crawling techniques is not practically possible. Another method was required.

It is also possible to find resources using a centralized bulletin board or registry that can act as an arbiter between things in the network. Such a solution can very easily use the HTTP protocol as it is defined. But it would requires knowledge of which bulletin board to use, and how to use it, and an agreement between all participants in the network to use it. Even though the required information to make this work cannot be provided by the HTTP protocol itself, it can be provided by underlying network technologies such as DNS and DHCP. But it will still require a new resource to be introduced into the network, a resource that could fail and that would need maintenance and configuration. For ad hoc networks, another solution was needed; a solution that did not include additional components apart from the devices that will find each other and the components forming the network itself.

Using UDP instead of TCP

The problem with using HTTP for finding new resources was that HTTP is based on TCP, and as such is done over connections between two endpoints that must know each other. UDP, on the other hand, does not have this limitation. In UDP it is possible to send a datagram to a multicast address. Anybody subscribing to messages sent to a multicast address will receive them, regardless of who the sender is. Such multicast address subscription functionality is provided by the **Internet Group Message Protocol** (**IGMP**) and is built into the network router or switch being used in the network. If the router or switch fails, the network fails anyway, so using multicast addressing does not introduce an additional component into the network that can fail separately.

By making a brief alteration to how HTTP is used, making it use UDP and multicast addresses instead of TCP and unicast addresses, it became possible to broadcast HTTP requests on the network to nobody in particular. Anybody understanding the request can also choose to respond. To make communication more efficient, responses can be sent to the unicast address used by the original sender of the request. Using multi-cast HTTP requests in this way, communication between things in the network becomes possible, even though they are not aware of each other from the beginning. Sometimes, HTTPU is used to denote HTTP over UDP, and HTTPMU to denote HTTP over multicast UDP, as is shown in the following stack diagram.



There are three important things to keep in mind when using HTTP over UDP. Firstly, UDP does not support connections, and so the delivery of datagrams, and hence also the order of datagrams, is not guaranteed as it would be over TCP. This makes sending large requests or responses over UDP non-trivial. Since packets are sent without any additional header information, content is restricted to the size of one datagram, which is about 64 kilobytes of information.

Secondly, sending a request openly, in the wild, may cause multiple devices to respond. The sender must handle the case where nobody or more than one responds to a given request. Last but not least is the problem of who is allowed to receive multicast information. If information is sent in the clear, anybody subscribing to datagrams sent to that particular multicast address will be able to receive it. The sender can limit the number of router hops a datagram can travel before reaching the receiver. By restricting the number of router hops possible, it is possible to restrict how far a datagram travels in the network. This number is called **Time to live** (**TTL**).

The Simple Service Discovery Protocol

Using the fact that devices can communicate with each other without knowing the IP addresses of each one, it was possible to create a new protocol based on HTTPU and HTTPMU that formalizes methods for finding devices in the network. The resulting protocol was named **Simple Service Discovery Protocol** (**SSDP**). SSDP introduces two new HTTP methods: `NOTIFY` and `M-SEARCH`. The `NOTIFY` method is used by a device or service to notify the network about its existence. In this way, a device can learn about the existence of another without actively searching for it. The `M-SEARCH` method, on the other hand, is used when a device actively searches for other devices or services in the network.

SSDP specifies what multicast addresses to use, to facilitate interoperability. In IPv4 networks, the address used is `239.255.255.250`. In IPv6 networks, four different addresses are used: [FF02::C], [FF05::C], [FF08::C], and [FF0E::C] for link-local, site-local, organization-local, and global multicasting capabilities, respectively, available in IPv6. The port number is also restricted to 1900 for most requests, even though this might vary in different applications. For instance, multicast event notification in UPnP, which is based on SSDP, is done on port 7900.

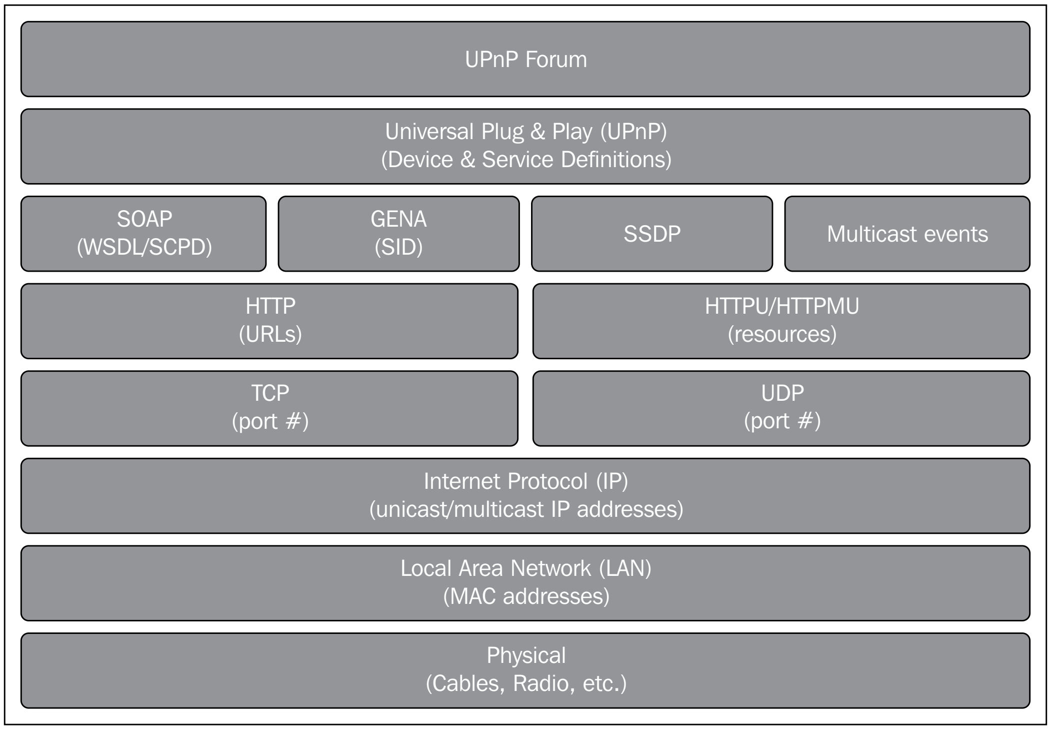
Handling events using GENA

As we discussed in *Chapter 2*, *The HTTP Protocol*, one of the problems with HTTP is how to handle asynchronous events. The method used in the previous chapter is to use delayed responses, where the server delays the response to a request until something happens. This method can be used even if the client resides behind a firewall. Another method that can be used in local area networks allows both parties to be clients of each other, allowing requests to be made in both directions. In this method, the original client subscribes to events of a certain type from the original server. This subscription is done in the form of a request/response pair, where the client registers its interest in events of the particular type.

At the same time, the client also provides a URL that can be used to send events to. When an event is triggered, the server becomes the client (in an HTTP sense) and sends the event to the original client (now a server). This process continues until the original client unsubscribes from event notification, or the subscription expires without having been renewed within a given time frame.

GENA extends normal HTTP (or HTTPU and HTTPMU, if UDP is desired) with two methods for subscription and un-subscription: the `SUBSCRIBE` and `UNSUBSCRIBE` methods, respectively. Subscriptions are maintained on each device and events are propagated to each subscriber using the provided call-back URLs.

The drawbacks of using this type of event handling are that all participants must reside in the same network, unprotected by firewalls. Another drawback is that all participants must implement both HTTP clients and HTTP server stacks. But, if SSDP is to be used to communicate with unknown devices, all parties need to have both client and server stacks implemented anyway, so adding GENA to the stack does not add a lot of code. And if the purpose of the solution is to provide event capabilities between devices in local area networks, such as in the home or in the office, GENA provides a suitable solution.



Tying it all together—UPnP

Using the previously mentioned protocols HTTP, HTTPU, HTTPMU, SSDP, and GENA, it is possible to create an architecture that enables devices to discover each other and what services they publish. Devices can also subscribe to events from each other, and asynchronous sending of events is also possible. All that needs to be done is to describe the details of how this is performed. This is exactly what UPnP does.

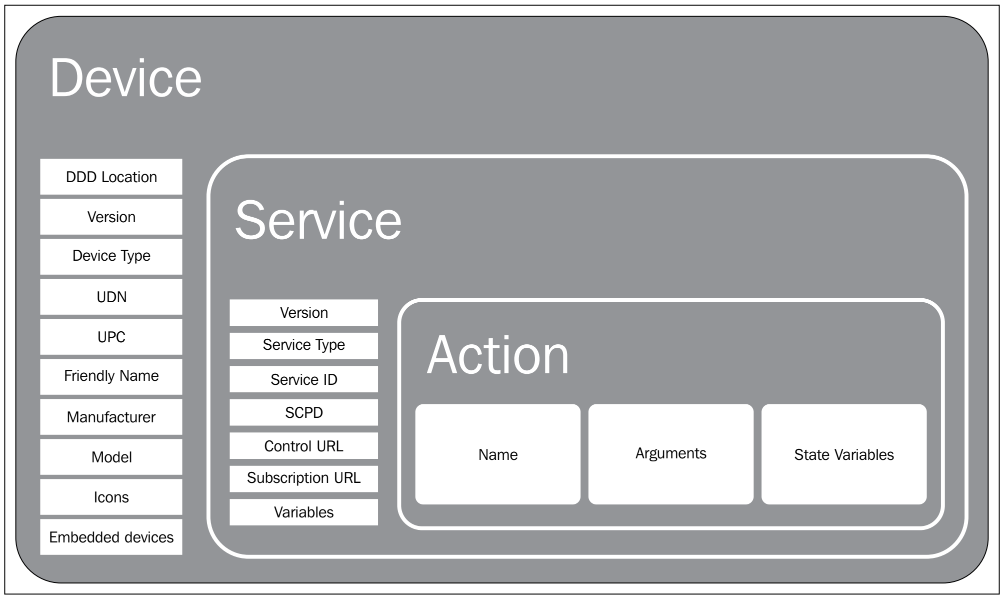
UPnP defines an object hierarchy for UPnP-compliant devices. Each device consists of a root device. Each root device can publish zero or more services and embedded devices. Each embedded device can iteratively by itself publish more services and embedded devices. Each service in turn publishes a set of actions and a set of state variables. Actions are methods that can be called on the service using SOAP web service method calls. Actions take a set of arguments. Each argument has a name, a direction (if it is input or output) and a state variable reference. From this reference, the data type of the argument is deduced. State variables define the current state of a service, and each one has a name, data type, and variable value. Furthermore, state variables can be normal, evented, and/or multicast-evented. When evented state variables change value, these values are propagated to the network through event messages. Normally evented state variables are sent only to subscribers using normal HTTP. Multicast-evented state variables are propagated through multicast HTTPMU NOTIFY messages on the SSDP multicast addresses being used, but using port number 7900 instead of 1900. There's no need to subscribe to such event variables in order to be kept updated on their values.

Simplifying the service architecture

The main invention of UPnP was the creation of a simplified device and service architecture that is easy to implement, simple to discover and automate, and at the same time versatile enough to avoid limiting device manufacturers in what they want to accomplish.

Each UPnP-compatible device in the network is described in a **Device Description Document** (**DDD**), an XML document hosted by the device itself. When the device makes its presence known to the network, it always includes a reference to the location of this document. Interested parties then download the document, and any referenced material, to learn what type of device this is, and how to interact with it. The document includes some basic information understandable by machines, such as UPnP version information, Device Type, **Universal Device Name** (**UDN**), and a **Universal Product Code** (**UPC**). It also includes information for human interfaces, such as a friendly name, manufacturer information, model information, and some graphical icons and links to a presentation web page for the device. Finally, the DDD includes references to any embedded devices, if any, and references to any services published by the device.

Each service published by a device is described in a stand-alone **Service Control Protocol Description** (**SCPD**) document, each one an XML document also hosted by the device. Even though SOAP is used to call methods on each service, UPnP-compliant services are drastically reduced in functionality compared to normal SOAP web services. SOAP and WSDL simply give devices too many options, making interoperability a problem. For this reason, a simpler service architecture is used. Instead of using WSDL to describe the service methods, the SCPD XML document does this directly.



Apart from a machine-readable UPnP version number, a Service Type, Service ID, Control URL, and an Event Subscription URL, the SCPD document basically consists of two lists: a list of actions (or web service methods) published by the service and a list of state variables managed by the service. State variables have a name, a data type, and a value. The SCPD document lists these attributes, and also whether the corresponding state variables are evented and/or multicast-evented. The number of possible data types is also reduced compared to SOAP, so only simple data types are included. The SCPD document completely avoids the complexity of describing web service method calls, as has to be done in normal WSDL, by simply saying that web service methods only have named arguments that correspond to the above-mentioned state variables. This means that data types in web method calls are limited to the data types available for state variables. Each argument also has a direction attribute, meaning each attribute can be either an input or an output argument of the method call. One output argument can also be assigned the return value of the action. The previous illustration displays a simplified overview of the UPnP device architecture.

UPnP Forum

To achieve any type of interoperability in networks, a standardization body is required. For UPnP, this standardization body is called UPnP Forum, and can be found on `http://upnp.org/`. UPnP Forum is an open organization and anybody can become a basic member. Apart from publishing documents about UPnP, the organization also defines a set of standardized device description documents and service control protocol description documents for various types of devices and their corresponding services. Some of these interfaces have become very popular, especially for use with consumer electronics and home entertainment. UPnP has also become a cornerstone of **Digital Living Network Alliance**, (**DLNA**). For more information you can visit `http://www.dlna.org/`.

The future of UPnP

UPnP is now undergoing important updates and additions to the original set of specifications. These updates go under the name of **UPnP+** and include three important additions: Interfaces for Internet of Things (or Sensor Management), interfaces for multi-screen applications, and the UPnP Cloud Annex, where UPnP will take the leap from local area networks into the cloud, using XMPP technology.

J
Data Types in UPnP

One of the major restrictions UPnP has made to service architecture is related to supported data types. UPnP only supports simple data types: integers, floating point numbers, strings, characters, Boolean values, date and time values, **uniform resource identifiers** (**URI**s), **universally unique identifiers** (**UUID**s), and two representations of binary data. This doesn't mean manufacturers cannot add custom data types to their specification files, but it means that, in order to be able to use common tools while expecting the device to be interoperable, you are limited to these data types. For developers, it is important that they know about these data types. The following table lists data types supported by **Universal Plug and Play** (**UPnP**):

Data type	**Description**	**.NET equivalent**
`ui1`	This is an unsigned 1-byte integer	`System.Byte`
`ui2`	This is an unsigned 2-byte integer	`System.UInt16`
`ui4`	This is an unsigned 4-byte integer	`System.UInt32`
`i1`	This is an signed 1-byte integer	`System.SByte`
`i2`	This is an signed 2-byte integer	`System.Int16`
`i4`	This is an signed 4-byte integer	`System.Int32`
`int`	This is an signed integer with an unspecified size	`System.Numerics.BigInteger`
`r4`	This is a 4-byte floating-point number	`System.Float`
`r8`	This is an 8-byte floating-point number	`System.Double`
`number`	This is the same as r8	`System.Double`

Data type	Description	.NET equivalent
`fixed.14.4`	This is the same as r8 but has no more than 14 digits to the left of the decimal point and no more than 4 to the right	`System.Double`
`float`	This is a floating-point number with an unspecified size	N/A
`char`	This is a Unicode character	`System.Char`
`string`	This is the Unicode string	`System.String`
`date`	This is the date in a subset of ISO 8601 format without time data	`System.DateTime`
`dateTime`	This is the date in ISO 8601 format with an optional time but no time zone	`System.DateTime`
`dateTime.tz`	This is the date in ISO 8601 format with an optional time and an optional time zone	`System.DateTimeOffset`
`time`	This is the time in a subset of ISO 8601 format with no date and no time zone	N/A
`time.tz`	This is the time in a subset of ISO 8601 format with an optional time zone but no date	N/A
`boolean`	This is a Boolean value ("0" or "1")	`System.Boolean`
`bin.base64`	This is a base64-encoded binary array of data	N/A
`bin.hex`	This is a hexadecimal representation of a binary array of data	N/A
`uri`	This is a Universal Resource Identifier	`System.Uri`
`uuid`	This is a Universal Unique ID	`System.Guid`

K
Camera Web Interface

To verify that the camera works, we create a simple web interface publishing a simple HTML page displaying an image, showing a picture taken by the camera. We register these two resources on our HTTP server as follows:

```
httpServer.Register ("/html", HttpGetHtmlProtected, false);
httpServer.Register ("/camera", HttpGetImgProtected, true);
```

We will later publish the same resources in our UPnP interface. The UPnP interface differs from the normal web interface in that the UPnP works in the unprotected local area network, while the web interface is supposed to be accessible from the Internet. This means we need a protected and an unprotected version of the same resource, which can be obtained by using the following code:

```
private static void HttpGetHtmlProtected (HttpServerResponse resp,
  HttpServerRequest req)
{
  HttpGetHtml (resp, req, true);
}

private static void HttpGetHtmlUnprotected (
  HttpServerResponse resp, HttpServerRequest req)
{
  HttpGetHtml (resp, req, false);
}
```

With the following code, we protect the web resource in the customary fashion, but only if required:

```
private static void HttpGetHtml (HttpServerResponse resp,
  HttpServerRequest req, bool Protected)
{
  networkLed.High ();
```

```
    try
    {
      if (Protected)
      {
        string SessionId = req.Header.GetCookie ("SessionId");
        if (!CheckSession (SessionId))
          throw new HttpTemporaryRedirectException ("/");
      }
```

Adding query parameters

We will make the web page customizable using query parameters in the URL. In this way, we can create links containing a certain configuration of the camera. To use query parameters, the following code is used:

```
LinkSpriteJpegColorCamera.ImageSize Resolution;
string Encoding;
byte Compression;

GetImageProperties (req, out Encoding, out Compression,
  out Resolution);
```

We do the parsing of the query parameters from a separate method, as shown in the following code, to be able to reuse it from other resources:

```
private static void GetImageProperties (HttpServerRequest req,
  out string Encoding, out byte Compression,
  out LinkSpriteJpegColorCamera.ImageSize Resolution)
{
```

We begin by extracting what image encoding to use. If not present, we use the default image encoding with the help of the following code:

```
if (req.Query.TryGetValue ("Encoding", out Encoding))
{
  if (Encoding != "image/jpeg" && Encoding != "image/png" &&
    Encoding != "image/bmp")
      throw new HttpException (
        HttpStatusCode.ClientError_BadRequest);
} else
  Encoding = defaultSettings.ImageEncoding;
```

Similarly, we extract the image compression level to use. If not provided, the next code snippet sets the image compression as default:

```
string s;
if (req.Query.TryGetValue ("Compression", out s))
{
  if (!byte.TryParse (s, out Compression))
    throw new HttpException
      (HttpStatusCode.ClientError_BadRequest);
}
else
  Compression = defaultSettings.CompressionLevel;
```

Finally, the desired image resolution is extracted. If not found, the default image resolution is used, as follows:

```
  if (req.Query.TryGetValue ("Resolution", out s))
  {
    if (!Enum.TryParse<LinkSpriteJpegColorCamera.ImageSize>
      ("_" + s, out Resolution))
        throw new HttpException (
          HttpStatusCode.ClientError_BadRequest);
  } else
    Resolution = defaultSettings.Resolution;
}
```

Generating the web presentation

The generation of the web page is simple and straightforward. We return an HTML page that contains an IMG tag referencing our `/camera` resource. We will forward any query parameters provided in the call to the camera resource, but we will always display the image in 640 x 480. This is shown in the following code:

```
resp.ContentType = "text/html";
resp.Encoding = System.Text.Encoding.UTF8;
resp.Expires = DateTime.Now;
resp.ReturnCode = HttpStatusCode.Successful_OK;
resp.Write ("<html><head/><body><h1>Camera, ");
resp.Write (DateTime.Now.ToString ());
resp.Write ("</h1><img src='camera?Encoding=");
resp.Write (Encoding);
resp.Write ("&Compression=");
resp.Write (Compression.ToString ());
resp.Write ("&Resolution=");
```

```
      resp.Write (Resolution.ToString ().Substring (1));
      resp.Write ("' width='640' height='480'/>");
      resp.Write ("</body><html>");
    }
    finally
    {
      networkLed.Low ();
    }
  }
```

Creating the image resource

As for the web resource method, the image resource method will be used both in our web interface, where it needs protection, and in the local network, where it doesn't. For this, we create the following two methods:

```
private static void HttpGetImgProtected (HttpServerResponse resp,
  HttpServerRequest req)
{
  HttpGetImg (resp, req, true);
}

private static void HttpGetImgUnprotected (
  HttpServerResponse resp, HttpServerRequest req)
{
  HttpGetImg (resp, req, false);
}
```

Protection is then done using the following code, but only if required:

```
private static void HttpGetImg (HttpServerResponse resp,
  HttpServerRequest req, bool Protected)
{
  networkLed.High ();
  try
  {
    if (Protected)
    {
      string SessionId =
        req.Header.GetCookie ("SessionId");
      if (!CheckSession (SessionId))
        throw new HttpException (HttpStatusCode.
          ClientError_Forbidden);
    }
```

We allow for customization of the resource by permitting input parameters in the query. We define two additional variables that will be used later on when transferring the image from the camera to the application:

```
LinkSpriteJpegColorCamera.ImageSize Resolution;
string Encoding;
byte Compression;
ushort Size;
byte[] Data;

GetImageProperties (req, out Encoding, out Compression,
  out Resolution);
```

Configuring the camera

Access to the camera has to be locked so that only one thread at a time accesses it. This is important to remember since web requests can come in from multiple requests. The `lock` method is used to lock the access to the camera:

```
lock (cameraLed)
{
  try
  {
    cameraLed.High ();
```

If the desired resolution is not the resolution currently configured, we need to reconfigure the resolution of the camera and reset it. If the reset procedure fails, which can happen, we will assume the camera is back to its default baud rate and we need to reset it to 115,200 baud. There is room here for more advanced error management. The resetting of the baud rate is done with the following code:

```
if (Resolution != currentResolution)
{
  try
  {
    camera.SetImageSize (Resolution);
    currentResolution = Resolution;
    camera.Reset ();
  } catch (Exception)
  {
    camera.Dispose ();
    camera = new LinkSpriteJpegColorCamera (
      LinkSpriteJpegColorCamera.BaudRate.Baud__38400);
    camera.SetBaudRate (
      LinkSpriteJpegColorCamera.BaudRate.Baud_115200);
    camera.Dispose ();
    camera = new LinkSpriteJpegColorCamera (
      LinkSpriteJpegColorCamera.BaudRate.Baud_115200);
  }
}
```

If the desired compression ratio differs from the current ratio, we reconfigure the camera accordingly, as follows:

```
  if (Compression != currentCompressionRatio)
  {
    camera.SetCompressionRatio (Compression);
    currentCompressionRatio = Compression;
  }
```

Taking the picture

With the following code, we ask the camera to take a picture and then download it. Notice that the camera always returns a JPEG image:

```
  camera.TakePicture ();
  Size = camera.GetJpegFileSize ();
  Data = camera.ReadJpegData (Size);
```

For now, we are done with the camera, so we release it for this time as follows:

```
      errorLed.Low ();
    }
    catch (Exception ex)
    {
      errorLed.High ();
      Log.Exception (ex);
      throw new HttpException (
        HttpStatusCode.ServerError_ServiceUnavailable);
    }
    finally
    {
      cameraLed.Low ();
      camera.StopTakingPictures ();
    }
  }
```

Encoding and returning the photo

We now have a binary-encoded image. But the image is JPEG-encoded. If another encoding is desired, we first need to re-encode the image as follows:

```
if (Encoding != "imgage/jpeg")
{
  MemoryStream ms = new MemoryStream (Data);
  Bitmap Bmp = new Bitmap (ms);
  Data = MimeUtilities.EncodeSpecificType (Bmp, Encoding);
}
```

Returning the image is then simple:

```
    resp.ContentType = Encoding;
    resp.Expires = DateTime.Now;
    resp.ReturnCode = HttpStatusCode.Successful_OK;
    resp.WriteBinary (Data);
  }
  finally
  {
    networkLed.Low ();
  }
}
```

L
Text Encoding on the Web

The available software for Internet of Things might not be as fault-tolerant as normal HTTP-based web applications. To avoid some complexities for the receiver, it is important to avoid returning data that can cause problems for the receiver. One such complication is the **byte order mark** (**BOM**) of text-based content.

Normally, encoding of text stored in files is done by prefixing the text content with a byte sequence called the BOM. For instance, Unicode (UTF-16) uses the sequences 0xfe and 0xff or 0xff and 0xfe, depending on the byte order of characters, to tell the receiver Unicode is used. UTF-8 uses 0xef, 0xbb, and 0xbf, for instance. Normally, this is invisible to end users since applications that load and save the text content interpret it for the user.

However, when returning text content via a web server, text-encoding is specified using the Content-Type HTTP header as well. Any BOM here might confuse the receiver, especially if the byte order mark implies a different encoding from what is specified by the HTTP header. Furthermore, should the client recognize the BOM and remove it or handle it as part of the actual string? It gets more complicated when transmitting XML over HTTP. Then, you have three possible encodings to handle: first the byte order mark, then the value of the Content-Type header, and finally the encoding attribute available in the XML header itself.

When returning dynamic text content, this is normally not a problem since dynamically generated strings do not have byte order marks. But if you return text content stored in files and treat them as pure binary files, this may become a problem if two or three of the encodings do not match.

To avoid problems when working with text files, first try to avoid returning them as binary files, as this might return the byte order mark as well and also confuse the receiver, especially if a different encoding is provided in the Content-Type header. If working with XML documents you return over HTTP, make sure the XML header encoding, if available, actually matches the encoding used when transmitting the document or avoid the encoding attribute in the header to obviate problems. When working with embedded resources that you return over HTTP, as in our examples, make sure you save them in a text editor where you can choose to save them without the byte order mark.

M
Sending Mail with Snapshots

The UPnP-enabled controller provides an environment where we know of all still image cameras in the network, we are subscribed to the corresponding services, and we know what encoding parameters they use. Let's now use this information to do something interesting in our control application; let's take photos from all available cameras in the network, when an alarm goes off.

Mail settings

Before we send anything, we need to know where to send it. For this, we create a `MailSettings` class whose objects we can persist in the object database. We will leave the details of this class to the reader, since we have done several similar classes earlier. (Code for the class is available when downloading the source code for this chapter.) We need an instance of the settings class in the application:

```
internal static MailSettings mailSettings;
```

During application initialization, we load the previous mail settings. If none, we create a new set of settings. The developer can put their own mail settings in the default values, to test the mail feature.

```
mailSettings = MailSettings.LoadSettings ();
if (mailSettings == null)
{
  mailSettings = new MailSettings ();
  mailSettings.Host = "Enter mailserver SMTP host name here.";
  mailSettings.Port = 25;
  mailSettings.Ssl = false;
  mailSettings.From = "Enter address of sender here.";
  mailSettings.User = "Enter SMTP user account here.";
  mailSettings.Password = "Enter SMTP user password here.";
```

```
    mailSettings.Recipient =
      "Enter recipient of alarm mails here.";
    mailSettings.SaveNew ();
  }
```

We also leave it to the reader to create a web interface where the mail settings can be edited.

Connecting to a mail server

When we have mail settings, we can set up our connection to a **Simple Mail Transfer Protocol** (**SMTP**) mail server. In the `Clayster.Library.Internet.SMTP` namespace, there exists a static class that helps us transmit mail, as shown in the following code:

```
SmtpOutbox.Host = mailSettings.Host;
SmtpOutbox.Port = mailSettings.Port;
SmtpOutbox.Ssl = mailSettings.Ssl;
SmtpOutbox.From = mailSettings.From;
SmtpOutbox.User = mailSettings.User;
SmtpOutbox.Password = mailSettings.Password;
SmtpOutbox.OutboxPath = "MailOutbox";
SmtpOutbox.Start (Directory.GetCurrentDirectory ());
```

In the last call to the `Start` method, we provide a folder where outgoing mail messages can be stored if they cannot be transmitted at once. If there are temporary connection problems, the message will get sent later.

Once started, we must also make sure we remember to terminate the SMTP outbox when the application closes. This is done as follows:

```
SmtpOutbox.Terminate ();
```

Starting mail generation

We want to send a mail when an alarm is triggered. To do this, we update the `ControlHttp` method, where the control command is issued to set the alarm. We have:

```
case 1:   // Update Alarm
  bool b;

  lock (synchObject)
  {
    b = lastAlarm.Value;
  }

  HttpUtilities.Get ("http://Peter:Waher@192.168.0.23/ws/?" +
    "op=SetAlarmOutput&Value=" + (b ? "true" : "false"));
```

To this, we start a thread if the alarm goes off. The thread will capture photos from available cameras and send a mail:

```
if (b)
{
  Thread T = new Thread (SendAlarmMail);
  T.Priority = ThreadPriority.BelowNormal;
  T.Name = "SendAlarmMail";
  T.Start ();
}
break;
```

The mail generation thread then starts preparing a HTML-formatted message. HTML-formatted messages also allow for the embedding of images.

```
private static void SendAlarmMail ()
{
  MailMessage Msg = new MailMessage (mailSettings.Recipient,
    "Motion Detected.", string.Empty, MessageType.Html);
  StringBuilder Html = new StringBuilder ();
  IUPnPService[] Cameras;
  int c;

  Html.Append ("<html><head/><body><h1>Motion detected</h1>");
  Html.Append ("<p>Motion has been detected while the " +
    "light is turned off.</p>");
```

We also make sure to get a snapshot of available cameras in the network:

```
lock (stillImageCameras)
{
  c = stillImageCameras.Count;
  Cameras = new IUPnPService[c];
  stillImageCameras.Values.CopyTo (Cameras, 0);
}
```

To manage what we need to do in this method, we will need the following additional variables:

```
List<WaitHandle> ThreadTerminationEvents=new List<WaitHandle> ();
Dictionary<string,string> VariableValues;
string Resolution;
string ContentType;
string Extension;
ManualResetEvent Done;
int i, j;
```

Designing our mail

For each camera, we will take three shots with 5 seconds between each, at least. Each camera will be handled by a separate thread and the results are placed in a table with three column. Each camera will be presented on a separate row in the table:

```
if (c > 0)
{
  Html.Append ("<h2>Camera Photos</h2>");
  Html.Append ("<table cellspacing='0' ");
  Html.Append ("cellpadding='10' border='0'>");

  for (i = 0; i < c; i++)
  {
    Html.Append ("<tr>");
```

If the application has received the resolution and encoding state variables from the camera, we will assume the camera is working:

```
lock (stateVariables)
{
  if (!stateVariables.TryGetValue (
    Cameras [i].Device.UDN, out VariableValues))
      VariableValues = null;
}

if (VariableValues != null &&
  VariableValues.TryGetValue ("DefaultResolution",
    out Resolution) &&
  VariableValues.TryGetValue ("DefaultEncoding",
    out ContentType))
{
```

We create three table cells for three photos from the camera. We will have to invent file names for the images that we will embed in the final mail. Embedded resources in mails are accessed using the `cid:<content id>` URI scheme.

```
Extension = MimeUtilities.GetDefaultFileExtension (ContentType);

for (j = 1; j <= 3; j++)
{
  Html.Append ("<td align='center'><img src='cid:cam");
  Html.Append ((i + 1).ToString ());
  Html.Append ("img");
  Html.Append (j.ToString ());
  Html.Append (".");
  Html.Append (Extension);
  Html.Append ("' width='");
  Html.Append (Resolution.Replace ("x", "' height='"));
  Html.Append ("'/></td>");
}
```

Note that we can generate the above without actually knowing the size of the final image, since the camera has already reported its default resolution to us.

Spawning photo collection threads

We will also spawn a new thread for capturing photos from the camera. To know when the thread is done, we create a `ManualResetEvent` object that the thread signals when it is done. We can send one object to the thread method. Since we want to send multiple variable values, we embed them all into one object array that counts as one object.

```
Done = new ManualResetEvent (false);
ThreadTerminationEvents.Add (Done);

Thread T = new Thread (GetPhotos);
T.Priority = ThreadPriority.BelowNormal;
T.Name = "GetPhotos#" + (i + 1).ToString ();
T.Start (new object[]{ i, Cameras [i], ContentType,
  Extension, Msg, Done });
```

If we don't have the named state variables, we will assume something is wrong and simply state that the camera is not available:

```
    } else
    Html.Append ("<td colspan='3'>Camera " +
      "not accessible at this time.</td>");
```

```
        Html.Append ("</tr>");
      }
    }
```

Finally, we end the HTML portion of the mail as follows:

```
    Html.Append ("</table></body></html>");
```

Waiting for threads to finish

Before we can send the mail, we need the photo collection threads to terminate. For each thread, we have created a `ManualResetEvent` object. We wait for all these to complete, but not for more than 30 seconds:

```
    if (ThreadTerminationEvents.Count > 0)
      WaitHandle.WaitAll (ThreadTerminationEvents.ToArray (),
        30000);
```

Each thread has now embedded its corresponding photos into the mail message. We put our generated HTML into the mail message body and send it:

```
      Msg.Body = Html.ToString ();
      SmtpOutbox.SendMail (Msg, mailSettings.From);
    }
```

Getting photos from the camera

The first thing we do in the thread that collects photos is to extract the parameters sent to it in the object array. This is done with the following code:

```
    private static void GetPhotos (object State)
    {
      object[] P = (object[])State;
      int i = (int)P [0];
      IUPnPService Service = (IUPnPService)P [1];
      string ContentType = (string)P [2];
      string Extension = (string)P [3];
      MailMessage Msg = (MailMessage)P [4];
      ManualResetEvent Done = (ManualResetEvent)P [5];
      DateTime Next = DateTime.Now;
```

Executing actions on a service is easy. A `Variables` collection is used to pass parameters to and from the action. We get the image URL for the camera, by calling the `GetDefaultImageURL` action as follows:

```
try
{
  UPnPAction GetDefaultImageURL =
    Service ["GetDefaultImageURL"];
  Variables v = new Variables ();
  GetDefaultImageURL.Execute (v);
  string ImageURL = (string)v ["RetImageURL"];
```

We then parse the URL of the image and open an HTTP connection to the camera:

```
ParsedUri ImageURI = Web.ParseUri (ImageURL);
HttpResponse Response;
int ms;
int j;

using (HttpSocketClient Client = new HttpSocketClient (
  ImageURI.Host, ImageURI.Port,
  ImageURI.UriScheme is HttpsUriScheme,
  ImageURI.Credentials))
{
  Client.ReceiveTimeout = 20000;
```

Before requesting a photo, we wait for the correct time, 5 seconds have to pass (at least) between photos:

```
for (j = 1; j <= 3; j++)
{
  ms = (int)System.Math.Round (
    (Next - DateTime.Now).TotalMilliseconds);
  if (ms > 0)
    Thread.Sleep (ms);
```

We then request the photo and embed it into our mail message. Note that we don't need to decode the image and re-encode it into the mail message. We already know the content type of the image and have it encoded accordingly.

```
  Response = Client.GET (ImageURI.PathAndQuery, ContentType);
  Msg.EmbedObject ("cam" + (i + 1).ToString () + "img" +
    j.ToString () + "." + Extension,
    ContentType, Response.Data);

  Log.Information ("Click.", EventLevel.Minor,
    Service.Device.FriendlyName);

  Next = Next.AddSeconds (5);
}
```

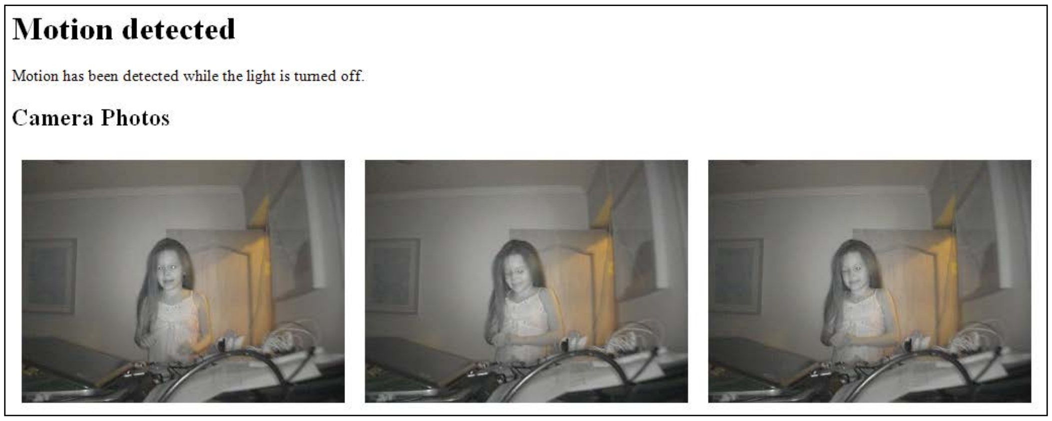
Signalling thread completion

When the photos have been collected, we close our connection and signal thread completion by setting the `ManualResetEvent` object passed to the thread. We make sure to log any exceptions that may occur and put the thread completion signal within a final statement, to make sure it is executed regardless of what happens in the method:

```
        }
      }
      catch (ThreadAbortException)
      {
        Thread.ResetAbort ();
      }
      catch (Exception ex)
      {
        Log.Exception (ex);
      }
      finally
      {
        Done.Set ();
      }
    }
```

Trying out the new camera

Now that we have four Raspberry Pi's running—one with a sensor, one with an actuator, one with a camera, and a controller controlling all three—we can put them to good use. For instance, we can see who it is that sneaks into our office and steals our resistors…


Motion detected

Motion has been detected while the light is turned off.

Camera Photos

N
Tracking Cameras

Now that we have a camera that publishes itself and its services using UPnP, we can track the available cameras in the network and their settings. This appendix shows how such tracking is done in the controller application.

Managing events using UPnP

As in our camera project, we need to create a UPnP interface for the network. To do this, we need an HTTP server and an SSDP client:

```
private static HttpServer upnpServer;
private static SsdpClient ssdpClient;
```

First, we set up the HTTP server; the camera project used a similar technique:

```
upnpServer = new HttpServer (8080, 10, true, true, 1);
Log.Information ("UPnP Server receiving requests on port " +
  upnpServer.Port.ToString ());
```

We also set up the SSDP client in a similar manner:

```
ssdpClient = new SsdpClient (upnpServer, 10,
  true, true, false, false, false, 30);
```

When the application closes, we need to dispose of these two objects to make sure any threads are closed as well. Otherwise, the application will not close properly:

```
ssdpClient.Dispose ();
upnpServer.Dispose ();
```

In our controller application, we will listen to notifications from the UPnP-compliant still-image cameras instead of actively publishing interfaces of our own. The SSDP client maintains a list of the found devices and interfaces for us. All we need to do is react to the changes in this list. We do this by adding an event handler for the `OnUpdated` event, as follows:

```
ssdpClient.OnUpdated += NetworkUpdated;
```

We will maintain three lists in our application to keep track of the available cameras. First is a list of still-image camera services available:

```
private static Dictionary<string, IUPnPService> stillImageCameras;
```

Then we need a list of subscriptions made to these services and when they expire:

```
private static SortedDictionary<DateTime, Subscription>
  subscriptions;
```

Lastly, we need a list of state variable values ordered by the unique device name of the device reporting them:

```
private static Dictionary<string,Dictionary<string,string>>
  stateVariables;
```

Reacting to network updates

In our `OnUpdated` event handler, we can examine the SSDP client, which contains a list of devices found in the network in the `Devices` property:

```
private static void NetworkUpdated (object Sender, EventArgs e)
{
  IUPnPDevice[] Devices = ssdpClient.Devices;
```

To detect which devices have been removed from the network, we first make a copy of those we have already registered in our application:

```
Dictionary<string, IUPnPService> Prev =
  new Dictionary<string, IUPnPService> ();

lock (stillImageCameras)
{
  foreach (KeyValuePair<string, IUPnPService> Pair in
    stillImageCameras)
    Prev [Pair.Key] = Pair.Value;
```

We then loop through all the devices found and all their services to see whether we can find a still-image camera service somewhere. If we do, we remove it from our list first. Those left in the list will have been removed from the network:

```
foreach (IUPnPDevice Device in Devices)
{
  foreach (IUPnPService Service in Device.Services)
  {
    if (Service.ServiceType == "urn:schemas-" +
      "upnp-org:service:" +
      "DigitalSecurityCameraStillImage:1")
    {
      Prev.Remove (Device.UDN);
```

If we detect a new device with a still-image camera service available, we add it to the list:

```
if (!stillImageCameras.ContainsKey (Device.UDN))
{
  stillImageCameras [Device.UDN] = Service;
  Log.Information ("Still image camera found.",
    EventLevel.Minor, Device.FriendlyName);
```

Subscribing to events

To be able to subscribe to events from the newly found service, we need to construct a callback URL to which the device can report any changes made to its **evented** state variables. To be able to create such a callback URL, we need to know which IP address the current application should report to. On a platform with multiple network adapters, we need to check which network adapter the device can be reached on. But since we run our controller on a Raspberry Pi, we can assume only one adapter is available. So we restrict our search to an IP address that uses the same IP protocol as the device does. We know which IP address the device uses by looking at its `Location` property, the URL of the device description document for its root device:

```
ParsedUri Location = Web.ParseUri (Device.Location);
string DeviceAddress = Location.Host;
System.Net.IPHostEntry DeviceEntry =
  System.Net.Dns.GetHostEntry (DeviceAddress);
```

We also need the IP addresses of the current machine:

```
string HostName = System.Net.Dns.GetHostName ();
System.Net.IPHostEntry LocalEntry =
  System.Net.Dns.GetHostEntry (HostName);
string LocalIP = null;
```

We will then see whether we can find a pair of IP addresses that match protocols:

```
foreach (System.Net.IPAddress LocalAddress in
  LocalEntry.AddressList)
{
  foreach (System.Net.IPAddress RemoteAddress in
    DeviceEntry.AddressList)
  {
    if (LocalAddress.AddressFamily ==
      RemoteAddress.AddressFamily)
    {
      LocalIP = LocalAddress.ToString ();
      break;
    }
  }

  if (LocalIP != null)
    break;
}
```

If an IP address match is found, we can create the callback URL and subscribe to events from the service. We request for a subscription of events for 5 minutes. The service can change this time period if it desires and return a **subscription identity** (**SID**), which we should store to be able to unsubscribe if needed. Note that the callback URL makes a reference to a resource named `/events`. We will define this resource a little later in this appendix. With the help of the following code, let's see how a callback URL, to which subscribed events will be sent is created:

```
if (LocalIP != null)
{
  int TimeoutSeconds = 5 * 60;
  string Callback = "http://" + LocalIP + ":" +
    upnpServer.Port.ToString () + "/events/" + Device.UDN;
  string Sid = Service.SubscribeToEvents (Callback,
    ref TimeoutSeconds);

  AddSubscription (TimeoutSeconds,
    new Subscription (Device.UDN, Sid, Service, LocalIP));
}
```

The `AddSubscription()` method will be discussed later in this appendix.

Unregistering cameras that have been removed

If cameras are removed from the network, it is important to update the internal structures representing the network to avoid memory leaks. After having looped through all the available devices and their services, any services still in our `Prev` list must be devices that have been removed since the last network update. We remove these from our list of cameras as well:

```
          }
        }
      }
    }

    foreach (KeyValuePair<string, IUPnPService> Pair in Prev)
    {
      Log.Information ("Still image camera removed.",
        EventLevel.Minor, Pair.Value.Device.FriendlyName);
      stillImageCameras.Remove (Pair.Value.Device.UDN);
```

We also remove any subscriptions we have registered on the device:

```
    foreach (KeyValuePair<DateTime, Subscription> Subscription in
      subscriptions)
    {
      if (Subscription.Value.UDN == Pair.Key)
      {
        subscriptions.Remove (Subscription.Key);
        break;
      }
    }
```

We also remove any state variables we have stored for the device. But we do this outside the current lock statement to avoid creating a deadlock:

```
        }
      }

      lock (stateVariables)
      {
        foreach (KeyValuePair<string, IUPnPService> Pair in
          Prev)
            stateVariables.Remove (Pair.Value.Device.UDN);
      }
    }
```

Remembering active subscriptions

When we have finally subscribed to events on the service, we need to store our active subscriptions together with the time when they expire. If we want to maintain the subscription, we need to update the subscription before the subscription expires. Any new subscriptions are stored in memory by calling the `AddSubscription` method. This method stores the subscription in the `subscriptions` dictionary, which is sorted on when the subscription expires. If multiple subscriptions expire at exactly the same time, random ticks are added until the expiry timestamp is unique:

```
private static void AddSubscription (int TimeoutSeconds,
  Subscription Subscription)
{
  lock (stillImageCameras)
  {
    DateTime Timeout = DateTime.Now.AddSeconds (
      TimeoutSeconds);

    while (subscriptions.ContainsKey (Timeout))
      Timeout.AddTicks (gen.Next (1, 10));

    subscriptions [Timeout] = Subscription;
  }
}
```

Creating the subscription record

The subscription class is a trivial class we've created to maintain information about an active subscription. It simply maintains information about four properties: the **unique device name** (**UDN**) of the device, an SID, the local IP address used in the callback URL, and a reference to the service object we subscribe to:

```
public class Subscription
{
  private string sid;
  private string udn;
  private string localIp;
  private IUPnPService service;

  public Subscription (string Udn, string Sid,
    IUPnPService Service, string LocalIp)
  {
    this.sid = Sid;
    this.udn = Udn;
    this.service = Service;
    this.localIp = LocalIp;
  }
```

These properties are then published in a read-only manner:

```
    public string UDN
    {
      get{return this.udn;}
    }

    public string SID
    {
      get{return this.sid;}
    }

    public IUPnPService Service
    {
      get{return this.service;}
    }

    public string LocalIp
    {
      get{return this.localIp;}
    }
  }
```

Checking active subscriptions

We need to make sure we update the existing subscriptions or they will be timed out on the corresponding devices, which will stop the flow of events. So we need to regularly check whether we get close to the expiry time of our existing subscriptions. In our `ControlHttp` method, we wait for control events to be issued to our actuator. We check for changes to LEDs and changes to the alarm status. This check is done using the following statement:

```
  switch (WaitHandle.WaitAny (Handles, 1000))
```

Here, `Handles` is an array of two event objects: the first represents the requirement to update the LEDs of the actuator and the second represents the requirement to update the alarm status. The wait times out after one second. In this `switch` statement, we can add a `default` statement that will execute if no control command is to be issued. This statement will therefore be called every second if no control command is to be issued. We will use this to control our subscriptions:

```
default:
  CheckSubscriptions (30);
  break;
```

In the `CheckSubscriptions` method, we go through all the active subscriptions to see whether there are any that will expire within the time frame specified in the `MarginSeconds` parameter:

```
private static void CheckSubscriptions (int MarginSeconds)
{
  DateTime Limit = DateTime.Now.AddSeconds (MarginSeconds);
  LinkedList<KeyValuePair<DateTime, Subscription>>
    NeedsUpdating = null;
  int TimeoutSeconds;
```

Note that we only need to loop through the dictionary, for as long as entries represent subscriptions expiring before the `Limit` margin time point, since the dictionary is sorted on the time when subscriptions expire:

```
lock (stillImageCameras)
{
  foreach (KeyValuePair<DateTime, Subscription> Subscription
    in subscriptions)
  {
    if (Subscription.Key > Limit)
      break;

    if (NeedsUpdating == null)
      NeedsUpdating = new LinkedList<KeyValuePair<
        DateTime, Subscription>> ();

    NeedsUpdating.AddLast (Subscription);
  }
}
```

Updating subscriptions

If there are subscriptions that need updating, we begin by removing them from our active subscription list:

```
if (NeedsUpdating != null)
{
  Subscription Subscription;

  foreach (KeyValuePair<DateTime, Subscription> Pair
    in NeedsUpdating)
  {
    lock (stillImageCameras)
    {
      subscriptions.Remove (Pair.Key);
    }

    Subscription = Pair.Value;
```

Then we try to update each subscription using the SID provided to us by the device. We also have to suggest a new timeout, which we set to 5 minutes. Remember that the suggested timeout can be updated by the device, if desired:

```
try
{
  TimeoutSeconds = 5 * 60;
  Subscription.Service.UpdateSubscription (Subscription.SID,
    ref TimeoutSeconds);
  AddSubscription (TimeoutSeconds, Subscription);
```

If the subscription update does not work, perhaps because the subscription has been forgotten or has timed out, we attempt to create a new subscription:

```
} catch (Exception)
{
  try
  {
    string Udn = Subscription.Service.Device.UDN;
    TimeoutSeconds = 5 * 60;
    string Sid = Subscription.Service.SubscribeToEvents (
      "http://" + Subscription.LocalIp + "/events/" +
       Udn, ref TimeoutSeconds);
    AddSubscription (TimeoutSeconds, new Subscription (Udn,
      Sid, Subscription.Service, Subscription.LocalIp));
```

If this fails as well, we try again in a minute if the device and service are still available in the SSDP client:

```
          } catch (Exception)
          {
            AddSubscription (60, Subscription);
          }
        }
      }
    }
  }
```

Receiving events

We have declared in our callback URL that a resource named `/events` will be used to receive events on. It is now time to define this resource. There is a predefined web resource called `UPnPEvents` in the `Clayster.Library.Internet.UPnP` namespace. We define such a static resource variable in our application:

```
private static UPnPEvents events;
```

We then create an instance and register it with our HTTP server dedicated to UPnP:

```
events = new UPnPEvents ("/events");
upnpServer.Register (events);
```

This resource has an event called `OnEventsReceived` that is raised whenever an event is received. We add an event handler for this event:

```
events.OnEventsReceived += EventsReceived;
```

In our event handler, we simply maintain the current state variable values sorted by the UDN of the device hosting the service reporting the event. This makes it easy to access them when necessary:

```
private static void EventsReceived (object Sender,
  UPnPPropertySetEventArgs e)
{
  Dictionary<string,string> Variables;
  string UDN = e.SubItem;

  lock (stateVariables)
  {
    if (!stateVariables.TryGetValue (UDN, out Variables))
```

```
      {
        Variables = new Dictionary<string, string> ();
        stateVariables [UDN] = Variables;
      }

      foreach (KeyValuePair<string,string> StateVariable
        in e.PropertySet)
          Variables [StateVariable.Key] =
            StateVariable.Value;
    }
  }
```

O
Certificates and Validation

When working with communication based on SSL or TLS using certificates, you need to make a decision on whether you want to validate the authenticity of these certificates or not. The default option should always be to validate them. If for some reason you cannot validate your certificates—perhaps because they are self-signed, you are debugging or testing code, or because it is not important to authenticate the other endpoint—and you only use SSL/TLS as a means to encrypt data, making it difficult to eavesdrop, you can choose to trust all certificates. In trusted mode, you can communicate with remote endpoints using SSL/TLS, even if the certificates are not valid.

However, in trusting a certificate, you lose one of the key aspects of using certificates in the first place: the ability to authenticate the remote endpoint. If you connect to a server and it presents a certificate, the certificate should be validated to make sure the server is who it claims to be. If the certificate is not valid, somebody might try to pretend to be the server, just to be able to extract your user credentials or other sensitive information. For this reason, trusting certificates should not be done in production, where real and sensitive data is used.

Normally, on PCs, certificate validation does not pose a problem, as long as self-signed certificates are not used. But on small devices, it might. When a certificate is validated, the certificate itself and its credentials (or subject) are validated, and also the entire chain of trust for the certificate. Each certificate is created by somebody, a **Certificate Authority** (**CA**). Each time a certificate is created, a reference to the creator is embedded into the certificate. So, when a certificate is validated, the issuer is contacted first to not only validate the issuer, but also to make sure the corresponding certificate has not been revoked. A certificate that has been compromised can be revoked, which means all certificates based on the revoked certificate will automatically become invalid. This allows applications to avoid being tricked by somebody creating false certificates using stolen CA certificates, for instance, if detected. If the issuer in turn has another issuer itself, the same process is continued until you come to a root CA. This certificate will be a self-signed certificate.

Certificates come in public and private versions. The public version can be shared with anybody and is the version that is sent to clients during SSL/TLS negotiation. It is used to encrypt data that is sent to the remote endpoint. The private part is maintained securely by the remote endpoint and can be used to decrypt data that is sent to it. Only the holder of the private version of the certificate can decrypt the communication sent to it. During certificate validation, the security level of the entire chain of certificates is no more secure than the security level maintained in protecting the private version of the CA certificates, including the root certificate.

In order for a root certificate to validate properly, as for any self-signed certificate, the public version has to be installed into the system so the operating system knows it is safe to trust. Normal PC operating systems come with most common root certificates installed. But the Mono and Raspberry Pi do not have the same support. So, to make certificates validate on the Mono on Raspberry Pi, you might have to install the corresponding CA certificates yourself.

[Care should be taken to only install proper CA certificates. Once a certificate is installed, all certificates issued by that certificate will also become valid.]

If you use the XMPP server at `thingk.me`, it uses a certificate issued by StartCom Ltd. To make the certificate at `thingk.me` validate, we need to install the StartCom Ltd. CA certificates into our Raspberry Pi. We start by downloading and installing the root certificate with the following commands:

```
$ wget  http://www.startssl.com/certs/ca.crt
$ sudo certmgr -add -c -m CA ca.crt
$ sudo certmgr -add -c Trust ca.crt
```

[We use the `sudo` prefix to give the command superuser privileges. It can be removed if you are logged in as the root.]

StartCom also uses an intermediate CA server certificate that we will download and install, as follows:

```
$ wget http://www.startssl.com/certs/sub.class1.server.ca.crt
$ sudo certmgr -add -c -m CA sub.class1.server.ca.crt
$ sudo certmgr -add -c Trust sub.class1.server.ca.crt
```

To verify that the certificates have been properly installed, we can issue the following commands:

```
$ sudo certmgr -list -c Trust
$ sudo certmgr -list -c –m CA
```

The `certmgr` command is a tool that comes with the Mono.

If the certificates have been properly installed, you can run the applications against `thingk.me` again, without having to trust the server certificate.

P
Chat Interfaces

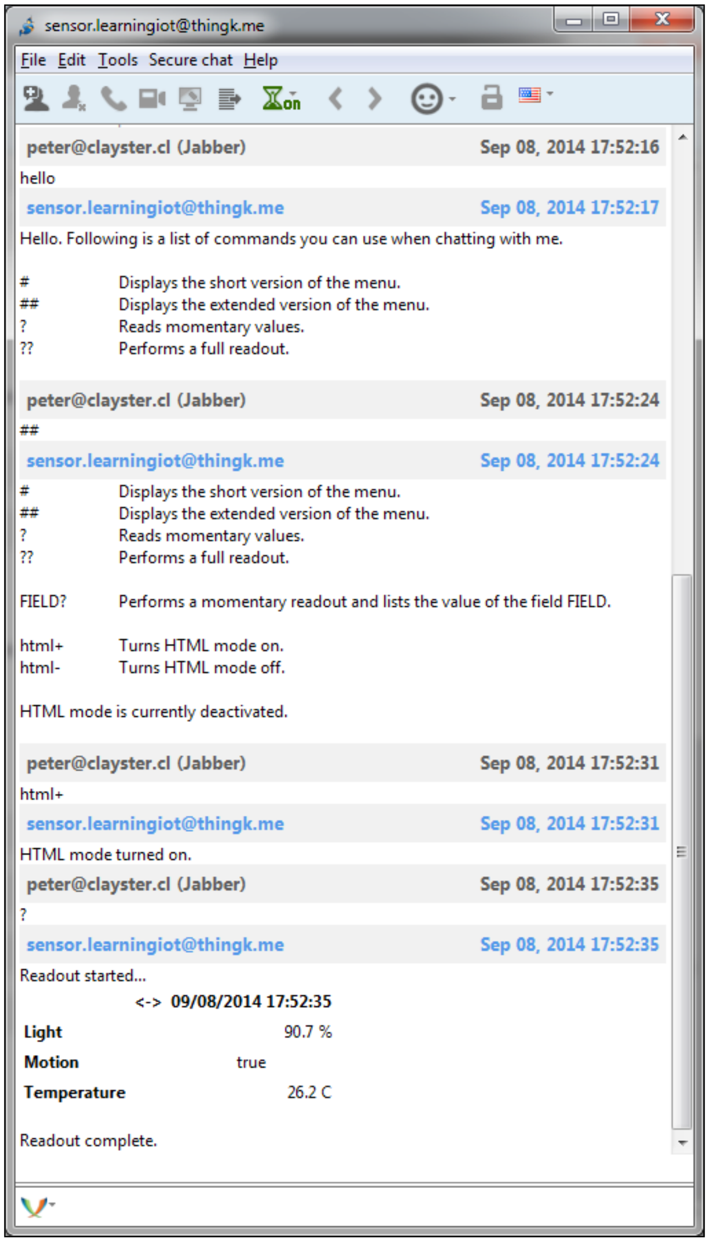
XMPP was originally developed for instant messaging in chat applications. To create an XMPP-enabled device that does not chat might not feel right. Furthermore, a chat application is a great human-to-machine interface that can be used for informative or debugging purposes. This appendix shows how simple it is to create a simple chat interface for a sensor or an actuator.

The `Clayster.Library.IoT.XmppInterfaces` namespace contains a class that implements a standard chat interface given an XMPP client and any sensor or actuator interfaces already defined. It then uses a standard syntax that is easy to remember and keeps the code to a minimum.

To create a chat server, just create an instance of the `XmppChatServer` class as follows. It takes a reference to the XMPP client and the provisioning server, and makes sure to negotiate all requests properly. Following are two optional parameters that can be null if no such interface is available. The first is a reference to a sensor server and the second is a reference to a control server.

```
xmppChatServer = new XmppChatServer (xmppClient,
  xmppProvisioningServer, xmppSensorServer, xmppControlServer);
```

The chat interface adapts itself to the capabilities of the optionally provided sensor and control interfaces. It checks what data fields and control parameters are available and who has the right to see and control them. The following screenshot illustrates an example chat session with our sensor:


sensor.learningiot@thingk.me
File Edit Tools Secure chat Help

```
peter@clayster.cl (Jabber)                         Sep 08, 2014 17:52:16
hello
sensor.learningiot@thingk.me                       Sep 08, 2014 17:52:17
Hello. Following is a list of commands you can use when chatting with me.

#        Displays the short version of the menu.
##       Displays the extended version of the menu.
?        Reads momentary values.
??       Performs a full readout.

peter@clayster.cl (Jabber)                         Sep 08, 2014 17:52:24
##
sensor.learningiot@thingk.me                       Sep 08, 2014 17:52:24
#        Displays the short version of the menu.
##       Displays the extended version of the menu.
?        Reads momentary values.
??       Performs a full readout.

FIELD?   Performs a momentary readout and lists the value of the field FIELD.

html+    Turns HTML mode on.
html-    Turns HTML mode off.

HTML mode is currently deactivated.

peter@clayster.cl (Jabber)                         Sep 08, 2014 17:52:31
html+
sensor.learningiot@thingk.me                       Sep 08, 2014 17:52:31
HTML mode turned on.
peter@clayster.cl (Jabber)                         Sep 08, 2014 17:52:35
?
sensor.learningiot@thingk.me                       Sep 08, 2014 17:52:35
Readout started...
                <-> 09/08/2014 17:52:35
Light                        90.7 %
Motion                  true
Temperature                  26.2 C

Readout complete.
```

Chat session with our sensor

Q
QR-Code

In our examples in this book, we use QR-code to transfer metadata information about a device to its owner. This appendix shows a simple way to generate and display such QR-code.

Generating a QR-code

A QR-code works like a two dimensional bar code and can encode a set of characters into a simple black-and-white image. The QR-code is easy to scan and decode, for instance using the camera on a smartphone. The inventor of the QR-code algorithm has given a free license to everybody to use it, and so QR-code has become a popular way of transmitting digital information using images. Instead of implementing code to generate QR-code, or using a specific library, we will use the Google Chart API, that can generate QR-code for us. All we need to do is send the string we want to encode, and it returns the corresponding image.

What we want to do is to transmit the metadata registered earlier to the owner in the form of a QR-code. This means we need to create a string containing the metadata that is simple to recognize and decode. The metadata is comprised of tags, string tags, and numerical tags. XEP-0347 stipulates that these tags are concatenated into a long string, delimited by the semicolon character ";". The string representation of a tag is formed by concatenating the tag name and the value, delimited by the colon character ":". If the tag is numerical, the tag name is prefixed by the hash sign "#". Finally, the entire string is prefixed by the string "IoTDisco;", so that it can be easily recognized. For our device, the final string might look something as follows:

```
IoTDisco;MAN:clayster.com;MODEL:LearningIoT-Sensor;
  KEY:acf9f815f5c64925843541db03d0f09e
```

Now that we have our string, getting a QR-code from Google Chart API is easy. The format of the URL for getting a small 48x48 QR-code is as follows:

```
https://chart.googleapis.com/chart?cht=qr&chs=48x48&
  chl=STRING_TO_ENCODE
```

> Note that we use the HTTPS protocol instead of the HTTP protocol to get the QR-code. We do this to avoid the possibility of eavesdroppers extracting the KEY tag value, which only the device and the owner are supposed to know.

We put our discovery string in the position of the `chl` query parameter value in the URL. We need to URL-encode it first, however. This will transform any hash characters to the character sequence %23. In our example, the final URL becomes:

```
https://chart.googleapis.com/chart?cht=qr&chs=48x48&
  chl=IoTDisco;MAN:clayster.com;MODEL:LearningIoT-Sensor;
  KEY:acf9f815f5c64925843541db03d0f09e
```

Entering this URL into a browser returns the following image, albeit much smaller:

QR-code containing metadata information

We code the preceding in a method called `RequestQRCode`, as follows:

```
private static void RequestQRCode ()
{
  Log.Information ("Loading QR Code.");
```

We use the HttpSocketClient class to create a connection to the Google Chart API:

```
using (HttpSocketClient Client = new HttpSocketClient (
  "chart.googleapis.com", HttpSocketClient.DefaultHttpsPort,
  true, HttpSocketClient.Trusted))
{
```

Note that we choose to trust the certificate from `chart.googleapis.com`. Even better would be not to trust it, but to validate it, to make sure nobody extracts your KEY information by pretending to be Google. This might require you to install some CA certificates, however. See *Appendix O*, *Certificates and Validation* for more information on how to do this.

Then we perform a `GET` operation to the resource:

```
HttpResponse Response = Client.GET ("/chart?cht=qr&chs=48x48& " +
  "chl=IoTDisco;MAN:clayster.com;MODEL:LearningIoT-Sensor;" +
  "KEY:" + xmppSettings.Key");
```

If the result is indeed a bitmap, we store it to avoid spamming the Google Image API server:

```
if (Response.Header.IsImage)
{
  xmppSettings.QRCode = (Bitmap)Response.Image;
  xmppSettings.UpdateIfModified ();
```

And if no owner has been registered, we display the QR code as well:

```
      if (string.IsNullOrEmpty (xmppSettings.Owner))
        DisplayQRCode ();
    }
  }
}
```

In commercial applications, libraries should be used to create QR codes, instead of using on-line APIs. This avoids the problem of sending metadata about the things you create to a third party.

Displaying the QR-code

Now that we have a QR-code, we need to display it somehow so the user can scan it and claim the device. In real life, such a QR-code might be placed on a sticker or shown on a display or similar. In our examples, we will simply show the QR-code in the Console window. It's a simple way to demonstrate the principle. We begin by defining a method for displaying our QR-code:

```
private static void DisplayQRCode ()
{
  Bitmap Bmp = xmppSettings.QRCode;
  if (Bmp == null)
    return;
```

We get the dimensions of the bitmap and loop though all its pixels:

```
ConsoleColor PrevColor = Console.BackgroundColor;
int w = Bmp.Width;
int h = Bmp.Height;
int x, y;

for (y = 0; y < h; y++)
{
  for (x = 0; x < w; x++)
  {
```

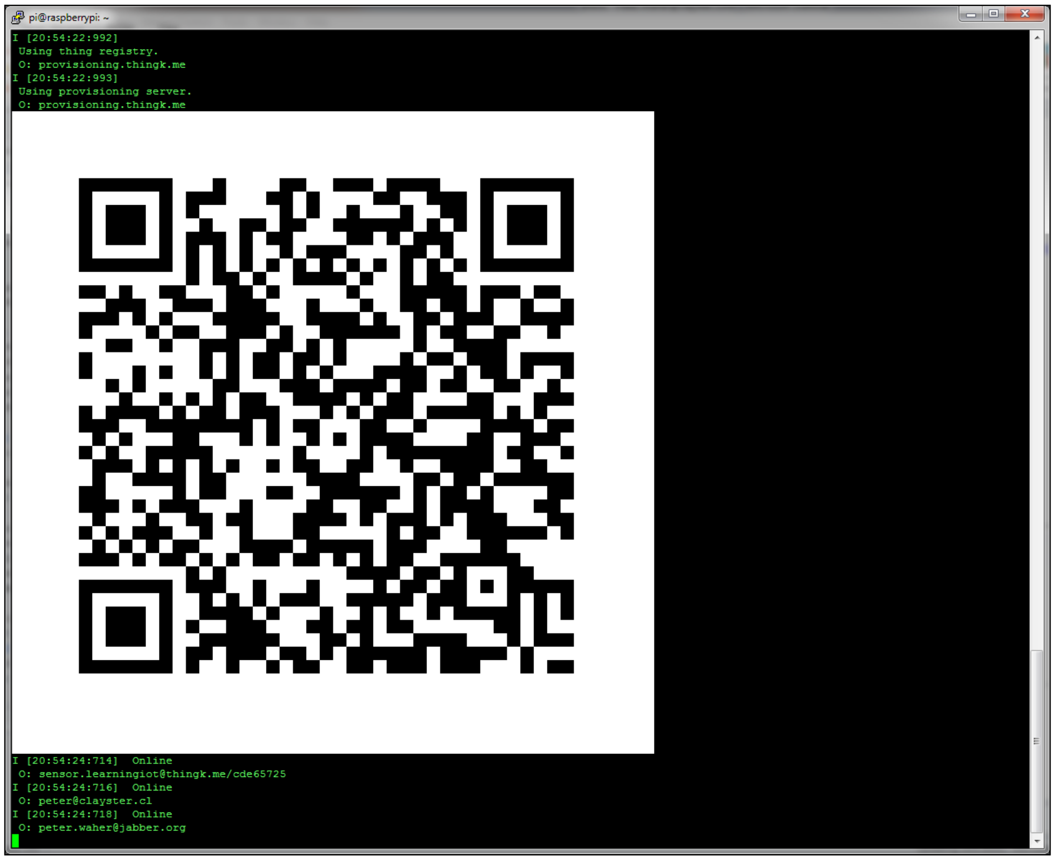
We output each pixel as a two-space character sequence. We choose two, for the simple reason that most terminal windows use a font where each character is roughly twice as high as it is wide.

```
  if (Bmp.GetPixel (x, y).B == 0)
    Console.BackgroundColor = ConsoleColor.Black;
  else
    Console.BackgroundColor = ConsoleColor.White;

  Console.Out.Write ("  ");
}
```

At the end of each line, we restore the original colors to make sure the rest of the line is not discolored if the terminal window scrolls:

```
    Console.BackgroundColor = PrevColor;
    Console.Out.WriteLine ();
  }
}
```

Testing our code in a console window will show something similar to the following image:



```
I [20:54:22:992]
 Using thing registry.
 O: provisioning.thingk.me
I [20:54:22:993]
 Using provisioning server.
 O: provisioning.thingk.me
```

```
I [20:54:24:714]  Online
 O: sensor.learningiot@thingk.me/cde65725
I [20:54:24:716]  Online
 O: peter@clayster.cl
I [20:54:24:718]  Online
 O: peter.waher@jabber.org
```

QR-code being output in a terminal window

Claiming ownership of the device

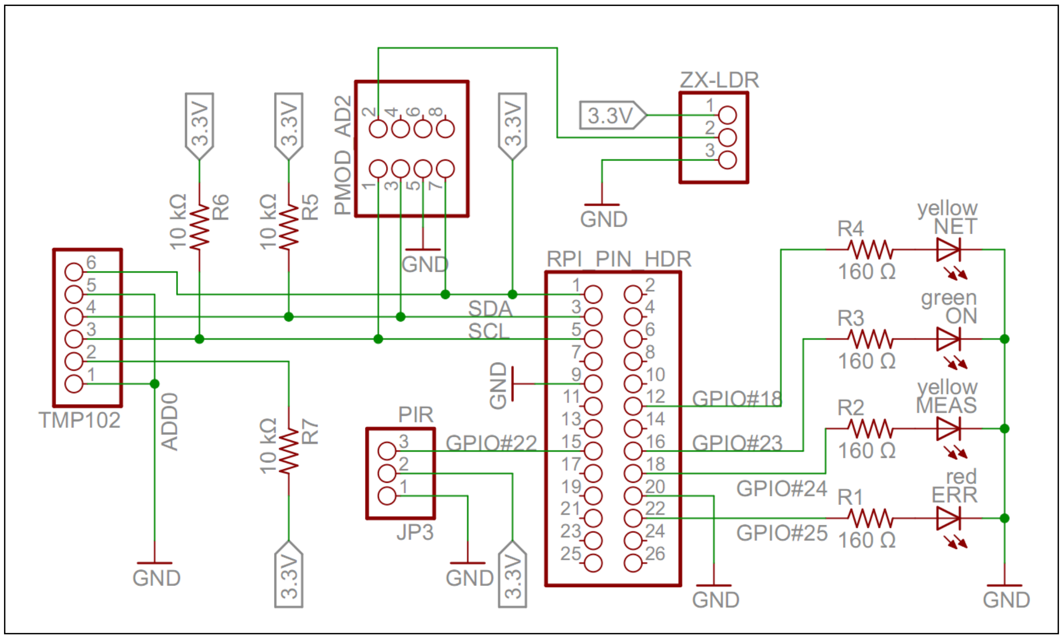
Once you have the QR-code, the owner can scan and decode it. The resulting string can be parsed to extract the metadata necessary to claim the ownership of the device. There are smartphone applications that let you do all this, using the camera on the phone. You can download one here: `https://www.thingk.me/Provisioning/Api.xml`.

R
Bill of Materials

This appendix contains a bill of materials for the boards used in the examples in this book.

The sensor

The circuit diagram used in our sensor project is as follows:

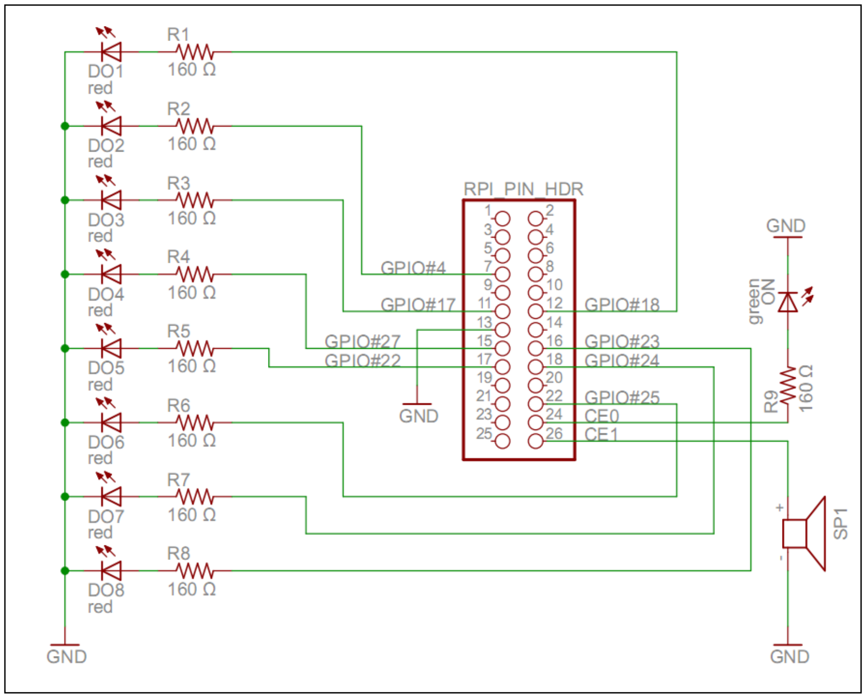


To build this circuit, we need the following components or variants thereof:

Part	**Value**	**Device Reference**	**Package**	**Description**
ERR	red	ZL-503RCA2	5 mm Round Type	LED Standard 20 mA
MEAS	yellow	ZL-503YCA2	5 mm Round Type	LED Standard 20 mA
NET	yellow	ZL-503YCA2	5 mm Round Type	LED Standard 20 mA
ON	green	ZL-504G0CA10	5 mm Round Type	LED Standard 20 mA
JP3	-	555-28027	1 x 03 Pins	PIR Motion Detector
PMOD_AD2	12 bit	PmodAD2	2 x 04 Pins	A/D converter
R1	160 Ω	CFA0207	Carbon Film	Resistors, Standard
R2	160 Ω	CFA0207	Carbon Film	Resistors, Standard
R3	160 Ω	CFA0207	Carbon Film	Resistors, Standard
R4	160 Ω	CFA0207	Carbon Film	Resistors, Standard
R5	10 kΩ	CFA0207	Carbon Film	Resistors, Standard
R6	10 kΩ	CFA0207	Carbon Film	Resistors, Standard
R7	10 kΩ	CFA0207	Carbon Film	Resistors, Standard
TMP102	-	TMP102	SOT563	Digital Temperature Sensor Breakout
ZX-LDR	-	ZX-LDR	1x03 Pins	Light Sensor Board
RPI_PIN_HDR	-	Raspberry Pi Model B	2x13 Pins	GPIO pin header, included with Raspberry Pi Model B
GPIO Ribbon	-	GPIO Ribbon	2x13 Pins	GPIO Cable Ribbon for Raspberry Pi

The actuator

The circuit diagram used in our actuator project is as follows:



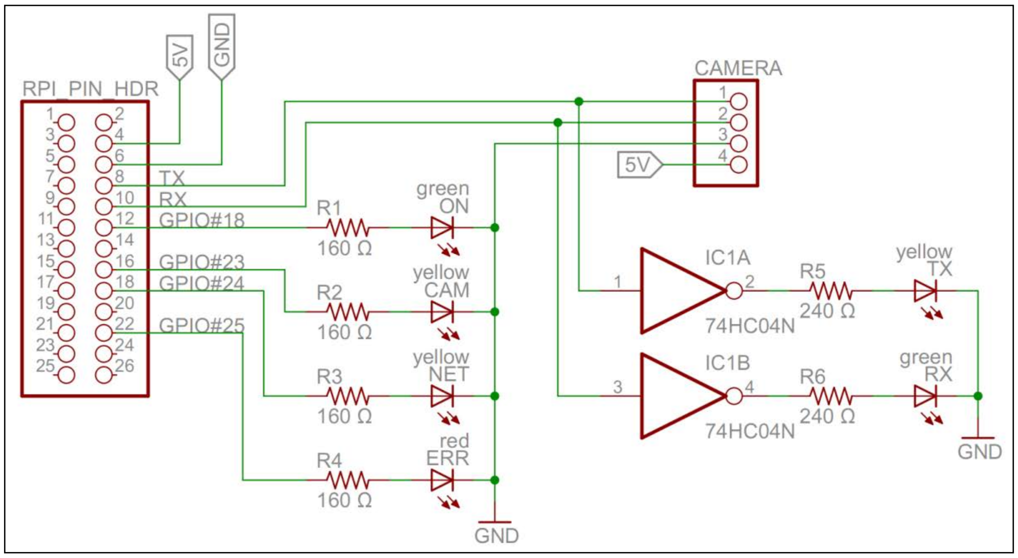
To build this circuit, we need the following components or variants thereof:

Part	Value	Device Reference	Package	Description
DO1	red	ZL-503RCA2	5 mm Round Type	LED Standard 20 mA
DO2	red	ZL-503RCA2	5 mm Round Type	LED Standard 20 mA
DO3	red	ZL-503RCA2	5 mm Round Type	LED Standard 20 mA
DO4	red	ZL-503RCA2	5 mm Round Type	LED Standard 20 mA
DO5	red	ZL-503RCA2	5 mm Round Type	LED Standard 20 mA
DO6	red	ZL-503RCA2	5 mm Round Type	LED Standard 20 mA
DO7	red	ZL-503RCA2	5 mm Round Type	LED Standard 20 mA
DO8	red	ZL-503RCA2	5 mm Round Type	LED Standard 20 mA
ON	green	ZL-504G0CA10	5 mm Round Type	LED Standard 20 mA
R1	160 Ω	CFA0207	Carbon Film	Resistors, Standard
R2	160 Ω	CFA0207	Carbon Film	Resistors, Standard

Part	Value	Device Reference	Package	Description
R3	160 Ω	CFA0207	Carbon Film	Resistors, Standard
R4	160 Ω	CFA0207	Carbon Film	Resistors, Standard
R5	160 Ω	CFA0207	Carbon Film	Resistors, Standard
R6	160 Ω	CFA0207	Carbon Film	Resistors, Standard
R7	160 Ω	CFA0207	Carbon Film	Resistors, Standard
R8	160 Ω	CFA0207	Carbon Film	Resistors, Standard
R9	160 Ω	CFA0207	Carbon Film	Resistors, Standard
SP1	-	GT-1005	-	Piezo Tweeter
RPI_PIN_HDR	-	Raspberry Pi Model B	2 x 13 Pins	GPIO PIN HEADERS, Included with Raspberry Pi Model B
GPIO Ribbon	-	GPIO Ribbon	2 x 13 Pins	GPIO Cable Ribbon for Raspberry Pi

The camera

The circuit diagram used in our camera project is as follows:



To build this circuit, we need the following components or variants thereof:

Part	**Value**	**Device Reference**	**Package**	**Description**
CAMERA	-	LinkSprite JPEG	1 x 04 Pins	Infrared color camera, Serial UART Interface
IC1	-	74HC04N	DIP14	Hex INVERTER
ERR	red	ZL-503RCA2	5 mm Round Type	LED Standard 20 mA
ON	green	ZL-504G0CA10	5 mm Round Type	LED Standard 20 mA
NET	yellow	ZL-503YCA2	5 mm Round Type	LED Standard 20 mA
CAM	yellow	ZL-503YCA2	5 mm Round Type	LED Standard 20 mA
RX	green	ZL-504G0CA10	5 mm Round Type	LED Standard 20 mA
TX	yellow	ZL-503YCA2	5 mm Round Type	LED Standard 20 mA
R1	160 Ω	CFA0207	Carbon Film	Resistors, Standard
R2	160 Ω	CFA0207	Carbon Film	Resistors, Standard
R3	160 Ω	CFA0207	Carbon Film	Resistors, Standard
R4	160 Ω	CFA0207	Carbon Film	Resistors, Standard
R5	240 Ω	CFA0207	Carbon Film	Resistors, Standard
R6	240 Ω	CFA0207	Carbon Film	Resistors, Standard
RPI_PIN_HDR	-	Raspberry Pi Model B	2 x 13 Pins	GPIO PIN HEADERS, Included with Raspberry Pi Model B
GPIO Ribbon	-	GPIO Ribbon	2 x 13 Pins	GPIO Cable Ribbon for Raspberry Pi